DIPLOMARBEIT

Software Development in the Environment of an International Research Organization

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines Diplom-Ingenieurs unter der Leitung von

Univ.Prof. Dipl.-Ing. Dr.techn. Hermann Kaindl
am
Institut für Computertechnik
Institutsnummer: 384

eingereicht an der Technischen Universität Wien
Fakultät für Elektrotechnik und Informationstechnik

von

Lukas Pilat
Matr.Nr. 0126113
Wilbrandtgasse 23, 1180 Wien

Wien, April 2011

Abstract

This work reports on an empirical case study about a software development project that was conducted at the European Organization for Nuclear Research (CERN). During the project, which lasted for eleven months, a new software package has been developed for a system that is part of CERN's particle accelerator complex. Empirical data about its progression was collected during the course of the project. This data was subsequently analyzed in form of a retrospective study.

Based on the experience from this project, the case study investigates the influence of domain knowledge on the software development process in general and on doing requirements in particular. Moreover, related issues regarding the communication between the project participants are taken into account. These are described together with the collected data, the results from its analysis, and the detailed context of the project.

The presented findings confirm that domain knowledge issues have a strong influence on requirements related activities. They suggest that a strictly sequential approach for software development can lead to project failure when developers lack essential domain-specific knowledge or cannot access it. Moreover, they challenge the applicability of document-driven approaches to software development under circumstances such as those encountered in this project. However, identifying key knowledge holders and fostering efficient knowledge transfer was found to be an essential factor for success in the project under study. Furthermore, the case showed the practical significance of selecting an appropriate form for representing requirements depending on the development practices in the given context.

Kurzfassung

Diese Arbeit beschreibt eine empirische Fallstudie über ein Softwareentwicklungsprojekt, welches in der Europäischen Organisation für Kernforschung (CERN) durchgeführt wurde. Während dieses elfmonatigen Projekts wurde ein neues Softwarepaket für ein System entwickelt, welches Teil der physikalischen Einrichtung von Teilchenbeschleunigern des CERN ist. Im Lauf dieses Projekts wurden empirische Daten über dessen Verlauf gesammelt. Diese Daten wurden anschließend in Form einer retrospektiven Analyse des Projektverlaufs untersucht.

Die auf die gesammelten Erfahrungen aufbauende Fallstudie untersucht den Einfluss von Domänenwissen auf den Softwareentwicklungsprozess im Allgemeinen und insbesondere im Hinblick auf Tätigkeiten zur Ermittlung der Anforderungen. Darüber hinaus werden Sachverhalte betreffend die Kommunikation zwischen Projektteilnehmern betrachtet. Diese sind im Zusammenhang mit den gesammelten Daten, der daraus resultierenden Analyse, sowie dem Kontext des Projekts im Detail beschrieben.

Die präsentierten Ergebnisse bestätigen den starken Einfluss von Domänenwissen auf Tätigkeiten aus dem Bereich der Anforderungsanalyse. Die Resultate legen nahe, dass ein strikter sequentieller Ansatz für die Softwareentwicklung zum Misserfolg eines Projekts beitragen kann, wenn den Softwareentwicklern essentielles Domänenwissen fehlt oder nicht zur Verfügung steht. Darüber hinaus stellen die Resultate die Anwendbarkeit von dokumentenbasierten Ansätzen zur Softwareentwicklung für Projekte mit vergleichbaren Umständen in Frage. Dagegen hat sich die bewusste Förderung von effizientem Wissensaustausch im Falle des gegebenen Projekts als maßgeblicher Erfolgsfaktor herausgestellt. Des Weiteren hat dieses Projekt die praktische Bedeutung der Auswahl der passenden Darstellungsform für Anforderungen entsprechend der vorherrschenden Entwicklungspraxis aufgezeigt.

Danksagung

Diese Arbeit konnte nur aufgrund der finanziellen Unterstützung meines Studiums durch meine Eltern entstehen, wofür Ihnen mein aufrichtiger Dank gebührt.

Die Diplomarbeit basiert auf einem Projekt, welches am CERN, der Europäischen Organisation für Kernforschung in Genf, durchgeführt wurde. Nur durch die Ermöglichung meiner Mitarbeit an dem Projekt konnte die Arbeit in dieser Form zustande kommen. Entscheidend zum Abschluss beigetragen hat auch die Bereitschaft von Herrn Professor Kaindl, diese Arbeit unter seiner Betreuung durchführen zu können.

Mein besonderer Dank gilt meiner langjährigen Freundin Xenia, die mir mit ihrer Fürsorge und ihrem Beistand auch durch die schwierigen Phasen des Studiums und der Diplomarbeit durchgeholfen hat. Für ihre Unterstützung und ihre Treue bin ich zutiefst dankbar.

Table of Content

1 INTRODUCTION

This work is about a software development project that has been conducted by the author during eleven months at the European Organization for Nuclear Research (CERN) in 2007 and 2008. During this project, the author contributed to the development of a software package for a specialized control system for one part of CERN's accelerator complex, which is Europe's largest research facility for particle physics.

Besides participating as the main software developer in this project, the author also conducted an empirical case study about the project. This case study and its findings are presented in this work. The case study focuses on factors that influenced the outcome and the progression of the project from a software-development point of view in this very specific context of an international research organization.

1.1 About the Project

The software development project was executed in the timeframe in which CERN completed its newest and largest addition to its particle accelerator facilities: the Large Hadron Collider (LHC) (see [98]). By the integration of this new accelerator into its existing infrastructure, many changes had to be made. Since most of CERN's particle accelerators are interconnected with each other, the changes affected the overall control systems of CERN [49] as well as the "LHC injector chain" [12].

It is within this context that a new software package for a subsystem of the "LINAC3" accelerator had to be redeveloped. This linear accelerator is one of the two particle beam production facilities for the LHC experiments. The LINAC3 accelerator, as part of CERN's "PS-Complex", produces the heavy lead ion particles, which are preprocessed by CERN's accelerator infrastructure before being injected into the LHC for experiments. These heavy ion experiments started to take place in 2010 [24].

The system for which the software was developed in this project was already in use for several years using a previously developed software package. For technical and organizational reasons that relate to the overall changes at CERN in conjunction with the "LHC era", it was decided that major parts of this software had to be redeveloped. Although the project had this reengineering characteristic, this case study shows that the project did not evolve as a reengineering project. The main goal was to develop new graphical user interface applications for the subsystem to be used by the operators controlling the LHC and its injector chain from the CERN Control Center (CCC) [73].

These aspects of the project's context introduce the main challenges it faced: the large amount of existing highly specialized and often proprietary technical infrastructure (hardware and software) into which the new software had to be integrated as well as the international and interdisciplinary characteristics of the working environment (mainly the different engineering approaches).

1.2 About the Case Study

The very specific context and environment of this software development project suggested conducting a case study that focuses on influence factors that might be especially pronounced in such a setting. As the project introduction points out, the large amount of highly specific and custom infrastructure, around which the new software had to be developed, indicated that *domain knowledge* would be an essential factor influencing the software development. As domain knowledge is tightly bound to requirements [42], an investigation of the requirements engineering approaches pursued in the project and its associated issues was of special interest. Further on, this suggested a closer look on potential communication issues between the project participants, especially in such an environment where people with very diverse background and engineering experience work together.

Those influence factors introduce the research topics of this case study. They show that the study was targeted more towards the "people oriented" factors than towards the purely technical aspects of software development. These "people oriented" and "human based" aspects of software development are a recurring topic in the software engineering literature [42, 6, 16, 21]. Especially with the emergence of the "agile" processes for software development [54], this field is getting new attention. Since also this project had aspects of agile development, the present study fits into this recent software engineering research context.

Major parts of this work are evolving around these different "people oriented" factors of software development. For this purpose, the author conducted qualitative and quantitative data collection during the course of the project. After the project's operational end, this data was processed and analyzed in a retrospective manner by the author. This constitutes the main content of this study. In addition, since this work also plays the role of describing the technical aspects of the software development effort conducted by the author, the descriptive

parts of this case study provide more technical details about the project than would have been necessary for the sole purpose of the case study.

1.3 Outline of the Work

The outline of this work was elaborated based on literature about empirical case studies. Especially the reference work of Yin [111] served as an important source for the case study methodology. The outline was elaborated by taking account of the recommendations found in [111 pp. 152-53] and [82 pp. 350-51]. It is composed of six chapters.

Following the introduction in this chapter, Chapter 2 discusses the role of case studies in software engineering, presents the methodology employed for the case study, describes the research topics, and gives an overview of the literature of the relevant context. Then, Chapter 3 is dedicated to the description of the environment and the technical context in which the project under study took place. This chapter also includes a description of the technical work done by the author during the project. Based on this context description, Chapter 4 presents the data collected during the project and the subsequent analysis conducted retrospectively after the end of the project. It discusses different findings of this analysis concerning the research topics of the study. These findings are further discussed and generalized into lessons learned in Chapter 5. Finally, a conclusion and outlook for further research in this domain are given in Chapter 6.

2 CASE STUDY METHODOLOGY

Case study reports, especially those targeted at academic audiences, are intended to make a study that has been conducted understandable and replicable, and should therefore dedicate an essential part to the description of the employed research methodology [111]. Accordingly, this chapter deals with the methodology of the reported case study.

Since the relevance of the case study as a research method in the software engineering domain is a recurring topic in the literature, this issue shall be discussed first in this chapter. Before treating the concrete approaches employed for the present study, it is necessary to define its very specific context, as this will reveal to be an influential factor. In relation to the context, it is important to clarify the research questions of the study and present the relevant literature. The last section of this chapter is dedicated to this detailed description of the design of the conducted study. It includes a discussion and detailed descriptions of the methods employed and the approaches taken for the collection and the analysis of the case study's evidence.

2.1 Case Studies in Software Engineering

In order to approach the topic of case studies in software engineering, it is first useful to overview how the case study as an empirical method is defined according to literature. Second, the role of case studies in the domain of software engineering shall be investigated by focusing on possible issues and its relevance as a research method in this discipline.

2.1.1 The Case Study as Empirical Method

A case study is originally one of several types of empirical methods for scientific research in social sciences [111 p. 1]. It has become a standard research method in sociology, medicine, psychology, and in economics [61 p. 54, 111 p. 1]. Its goal is "[…] to understand complex social phenomena […]" [111 p. 2] and "to retain the holistic and meaningful characteristics of real-

life events […]" [111 p. 2]. As every other empirical method, it is "[…] a test that compares what we believe to what we observe." [82 p. 347].

The case study method is one of several common empirical methods, which differ mainly by their scope. A good overview of the main characteristics of those common empirical methods is given by [61, 110] and can be summarized as:

- **Experiments:** "[…] they are concerned with a limited scope and most often are run in a laboratory setting. They are often highly controlled […]" [110 p. 9]. Through this high degree of control, experiments are particularly qualified for reproducibility. In general, their scope can be described as "research-in-the-small" [61 p. 53].
- **Case studies:** "Case study research is an observational method, i.e. it is done by observation of an on-going project or activity." [110 p. 10] "Data is collected for a specific purpose throughout the study. Based on the data collection, statistical analyses can be carried out. The case study is normally aimed at tracking a specific attribute or establishing relationships between different attributes." [110 p. 9] The scope of case studies can be described as "research-in-the-typical" [61 p. 53].
- **Surveys:** A survey "[…] is often an investigation performed in retrospect […]". "[…] it is possible to send a questionnaire to or interview a large number people covering whatever target population […]" [110 p. 10]. Thus, the scope of surveys can be described as "research-in-the-large" [61 p. 53] and research-in-the-past [110 p. 10].
- **Post-mortem Analysis:** Moreover, [110] cites the "post-mortem analysis" as another method, which is similar to what [111] calls the "history" method. It is always done in retrospective and it "[…] may be conducted by looking at project documentation […] or by interviewing people […]" and its scope can thus be described as "research-in-the-past-and-typical" [110 p. 10].

The main advantage of case studies is that they are very flexible. They allow investigations within the "real-life context" without the need of an external "control" [111 p. 13] and can be conducted retrospectively based on collected data [83]. Because of these advantages, this method was found to be very suitable for the present study.

In contrast, one of the main disadvantages of the case study methodology is that it poses more difficulties than the other methods in the interpretation of the collected data and in the deduction of general findings out of the single case that has been observed [61 p. 53]. Another difficulty becomes evident as soon as one starts looking for literature on how exactly to conduct such a study: As [61 p. 54] puts it: "[…] there is little formal documentation available on how to perform a proper case study […]". A reason might be that "[…] unlike formal experiments and surveys, case studies do not have a well-understood theoretical basis." [61 p. 52].

Despite the lack of specificity on how to conduct a case study and its apparent flexibility, the case study method is not just a buzzword for any kind of analysis about a typical case that has been observed. As reported in [83]: "A case study is not an exemplar or case history" and "is

not an experience report". On the contrary, the case study method is to be understood as a well-defined and comprehensive research strategy [83, 111 p. 14]:

> "[…] the case study as research strategy comprises an all-encompassing method – covering the logic of design, data collection techniques, and specific approaches to data analysis." [111 p. 14]

2.1.2 Issues of Empirical Methods in Software Engineering

Empirical research in general and case study research in particular is not as common and as widely recognized in the software engineering domain as it is in social sciences, psychology or in business [111 p. 1, 99 p. 557, 61 p. 54, 82 p. 347, 110 p. 18]. Moreover, every discipline has its own approaches for performing case studies [83]. This leads to the fact that there is no widespread standard for conducting proper case studies in the software engineering domain [61 p. 54]. Consequently, one of the main risks of the case study method is its potential to be taken as a loosely defined concept and not as a rigorous research strategy. Unfortunately, this risk seems to be poorly managed: According to [60 p. 721] "[…] the standard of empirical software engineering research is poor."

There were many case studies undertaken in software engineering – [18], [89], [88], [71], or [87] to cite just some of them – but only few literature exists on how to perform such a study in this domain [6 p. 479]. There is some literature about case studies in social sciences, which is at least in some parts general enough to be applied by its main principles to other research domains. Notable textbooks of this kind are [111] and [103], from which [111] was used as a main source of methodological guidance for the present study.

In order to improve the situation, a couple of authors have made contributions that focus directly on improving the quality of case studies in software engineering. Those efforts include for example guidelines [61, 60], a tutorial [83], a roadmap [82], and checklists [57]. This demonstrates the awareness of the scientific community about the issues of empirical research (and case studies in particular) in software engineering and at the same time, it shows the commitment towards higher quality standards. Although this should not mean that only "perfect" empirical studies are of scientific interest and only such will be accepted: "[…] what's important is not whether the study is textbook perfect, but whether the study and its conclusions taken as a whole are credible." [82 p. 349].

To give an example of why it is so difficult to come up with a unique standard for case studies in software engineering, it is interesting to look at a specific type of case studies that emerged in the software engineering domain: case studies for the evaluation of tools or methods. This type of case studies is important because it aims at resolving the often recurring question "which is better" [61 p. 54]. This contrasts sharply with the research questions "why" and "how" of classic case studies [111] and shows the wide-ranging variety of empirical research methods. With this diversity, it is understandable that establishing common standards is a difficult task.

In such a heterogeneous field, what does it mean for a case study to be "credible"? Despite the fact that there is no standard, it is still possible to identify a set of common good practices out of the most influential papers [61, 82, 99]. Those should be part of every proper case study in the software engineering domain and should thus make it more "credible". Those good practices are:

- Proper specification of the research questions
- Adequate preparation and description of the study method and its operational context
- Appropriate data collection during the study using well-defined methods
- Rigorous data analysis
- Drawing conclusions with respect to the initial research questions

A comparison with [111] shows that these elements correspond relatively well to the same basic principles of case studies in other research domains. This leads to the conclusion that the case study research in software engineering should follow the common good practices of case studies in general but should still be adaptable to special situations that might occur only within this specific domain (like the case studies for method or tool evaluation cited above). This very diversity of application of the case study research strategy is made evident by [111]:

> "[…] case studies can be conducted and written with many different motives, including the simple presentation of individual cases or the desire to arrive at broad generalizations based on case study evidence […]." [111 p. 15]

Since it is difficult to establish a standard of how a case study in software engineering should be done, it can be helpful to look at the opposite side and list the common "bad practices". They show what should be avoided in case studies. Especially [82 p. 349] gives a good overview of the most important deficiencies of empirical studies and particularly of case studies in software engineering:

- "There are too many papers whose only selling point is that they have lots of data. Data is not enough. […] data should be used to answer questions, not just to fill graphs."
- "[…] many empirical studies simply lack hypotheses. They pose no questions, they serve no well-defined end. Thus at the end of the study the researcher can only present observations about the data."
- "[…] the most important part of doing an empirical study is drawing conclusions. Many papers fail to do anything with their results. We need to learn something from every study and relate these things to theory and practice."
- "Too many empirical studies study the obvious. As this sometimes shows that the obvious isn't so obvious […]"

2.1.3 Importance of Case Studies in Software Engineering

In order to contrast the problems of empirical studies in the software engineering domain discussed in the previous section, it should be acknowledged that especially case studies are

nowadays an "important research methodology for software engineering" [57 p. 479] and are, despite the controversy, recognized by the research community [99 p. 557]. The method is very well suited for research in the software-engineering domain because of the human factor of the discipline [110 p. 7]. This non-technical aspect of software engineering is essential:

> "Certainly, software development is characterized by specific issues and problems. Still, we cannot forget that software development is carried out by teams of people involved in a highly creative activity. *It is, indeed, a human-centered process as many others engineering and design processes in our society.*" [50 p. 32]

This explains the importance of research methods coming originally from the sociological domain. A good explanation is given by [57 p. 479]:

> "Software engineering is a field of applied research. Research often involves investigating how people work in teams and projects in large organizations aiming to develop software."

Furthermore, [6 p. 443] shows why empirical research is not only *suitable* for the software engineering discipline but why it is a *necessity*:

> "We cannot rely solely on observation followed by logical thought. Software engineering is a laboratory science. It involves an experimental component to test or disprove theories, to explore new domains. We must experiment with techniques to see how and when they really work, to understand their limits, and to understand how to improve them."

Not only empirical research in general is needed but also case studies in particular are necessary because they are *the* instrument that allows empirical research in the real-life settings in which software engineering is practiced:

> "But where are the laboratories for software engineering? They can and should be anywhere software is being developed. Software engineering researchers needs industry-based laboratories that allow them to observe, build and analyze models." [6 p. 445]

> "I believe that we should observe professional software developers in their organizations. We should study what they do, and then build productive theories (a theory that addresses some important and significantly defensible explanation for the phenomena) to explain why they are doing what we observe." [108 p. 93]

Especially the case study methodology can deliver exactly what is needed. It is the method for "research-in-the-typical" [61 p. 53] because it analyzes what happens in real-life projects. As a conclusion to this topic, an influential paper on research in software engineering in general [48 p. 87] sums up:

> "Evaluative research must involve realistic projects with realistic subjects, and it must be done with sufficient rigor to ensure that any benefits identified are clearly derived from the concept in question. This type of research is time-consuming and expensive and, admittedly, difficult to employ in all software-engineering research."

2.2 Context and Research Questions

The central part of this case study is the very specific context under which the project under study took place. The following section gives an overview about the constellation of the project and demonstrates its relevance as a study object. Then, it is explained how the concrete research questions for the case study were elaborated. Finally, research literature surrounding the research topics is presented.

2.2.1 Operational Context of the Study

This section briefly introduces the software development project investigated in this case study, delineates its context and lists its essential aspects. The information presented here is intended to give an understanding of the relevant research topics and to show the relevance of the project as a study object. A detailed description of the project and its context can be found in Chapter 3.

This case study investigates a software development project, which lasted for eleven months. The work done by the author in the scope of this study incorporated two parts: The first part was the participation of the author as the main full-time software developer in the project conducted at CERN. The second part was the realization of the case study about that project from a software engineering point of view.

The first part consisted in executing the full software development project by gathering requirements, going through design, implementation, and testing until the handover of the final software "product". As the details about the project described in chapter 3 and 4 show, the author played a major role in all the different project related activities. This allowed a privileged insight into the course of the software development phases undertaken from the beginning until the end of the project. The second part consisted of all steps necessary to conduct a proper case study as suggested by [82, 111]. Those steps were the preparation of the research questions, the data collection during the project, the subsequent coding and analysis, and finally the interpretation of the findings.

The work of the author in both fields – the active project participation as well as the data collection for the case study – enabled a unique research constellation by allowing a thorough analysis from an "insider" point of view. This allowed a degree of insight that would have been difficult to acquire employing classical external participant observation [99 p. 558]. Evidently, the employed approach eliminates the possible bias of an external observer and instead, introduces the bias of the "insider". Several precautions were taken to benefit from the insider point of view and at the same time to minimize the possible bias as far as possible.

A major factor for the operational context of this study is the specific environment in which the project took place. CERN is the world's largest international research organization for particle physics [31] and as such, it constitutes a very specific environment for software development projects. It is important to point out that, although the project was conducted at a research institution, its context has characteristics comparable to industry. This can be explained by the fact that one of the main businesses of CERN is not only the research itself, but the conception, construction, and maintenance of its physical research facilities, which is reflected by CERN's organizational structure [29] as well as by its staffing. In 2008 for example, 81% of CERN's 2400 permanent staff members were applied scientists and engineers, technical staff and manual workers for building up and maintaining CERN's machines and their infrastructure, whereas only 3% were research physicists [25 p. 48]. From this point of view, CERN can be seen as an organization producing an operating research infrastructure that can be used by researchers from CERN or other institutions for physical measurements and experiments. This service provided by CERN can be seen as the organization's "product". The engineering work performed at CERN is therefore comparable to industry. This is exactly the frame within which the project under study took place. As a software development project, it was fully within the *engineering* activities of CERN and not within its *research* activities.

As can be seen through the aspects of the project detailed in chapter 3, projects similar to the one studied here can certainly be found in industry and the lessons learned from this case study can thus also be applied outside of the "research world". The comparable aspects are illustrated in the following.

The project under study took place within the *Controls Group* of the former *Accelerators and Beams Department* of CERN. An important aspect of the project's context is that CERN is often restructuring its internal organization [27, 28, 29], which is also comparable to what companies in industry do. Restructuring is exactly what led to the project under study: The responsibility for a special piece of custom-made hardware and its corresponding software was found to be in a section of the *Controls Group* where it did not belong, since the main business of that section was a completely different one.

CERN was founded in 1954 [31, 9] and since that time it continuously conceived and constructed new particle accelerator facilities and related technical infrastructure. Some of the technical equipment that is still in use today was designed several decades ago. This leads to the necessity of regular upgrade, maintenance, and modification projects in order to keep the existing machines fully functional within an evolving infrastructure [43]. The project under study fits into this context: Its goal was the development of a new software package for a subsystem of CERN's LINAC 3 accelerator whose construction began in 1990 [30]. The project was related to a part of the accelerator that was added only in 2003 [93]. However, following a change in the middleware software of the controls infrastructure of CERN's accelerator complex, the initial software package for this specific accelerator part had to be changed. This context of the project can be qualified as "maintenance" since parts of an existing system had to be adapted to changed requirements. This aspect of the project's

context is comparable to other projects in industry where maintenance of legacy systems caused by changing requirements is common [77 p. 43, 75].

One aspect of the project's context comes from the fact that it was conducted within a research environment and not a production or sales oriented environment. This is emphasized as CERN has the status of an international research organization fully financed by its member states [36] and has therefore no direct financial benefit from its engineering activities. Nevertheless, there is a restricted funding for all CERN projects based on the yearly budget approved by the CERN Council [36]. This aspect of the environment is comparable to Research and Development (R&D) departments of industry companies [95]. The environment is furthermore characterized by high heterogeneity of the engineering staff coming from different countries, cultures and educational backgrounds.

The last aspect of the context is that projects at CERN have often to deal with a big amount of "legacy". Many hardware and software systems constructed at CERN are custom developments conducted internally. Such development efforts have an inherent tendency to lack proper documentation that would facilitate future maintenance and reengineering efforts. This is also common in industry [10]. The project under study is a good example illustrating this aspect. It is interesting to note on this subject that exactly this need for easier documentation and knowledge exchange about legacy systems led to the invention of the World Wide Web at CERN [11, 14]. The specific domain knowledge necessary to deal with such legacy systems is one of the essential aspects of this study as will be shown by the research questions.

To sum up, the following important aspects of the project were identified:

- Software project within an international research organization
- Research context but still an engineering project comparable to industry
- Project with unclear responsibilities due to organizational restructuring
- Adaptation to a changed infrastructure
- Legacy system issues

Those aspects have influenced the direction of the case study conducted about this project.

2.2.2 Research Questions

The software development project investigated for this case study was given by the constellation in which the author came to work at CERN. The project and its context were therefore fixed and could not be influenced or changed in any significant way for the purpose of the case study.

Since the project and its context were imposed, the research questions driving this case study were intentionally left vague at the beginning. They were more research "topics" than concrete research "questions" and were derived from the specific aspects of the project and its context as described in section 2.2.1. These initial research topics were intended to guide the data collection process during the project's operational phase. Given the constellation of the

project, it was anticipated that interesting issues would arise during the course of the project and that they would lead towards more concrete research questions.

Analyzing the characteristics of the given project and its context presented in 2.2.1 led to the following preliminary research topics: communication with stakeholders, requirements engineering, and software development process. The following list explains these research topics and shows their relevance for the given project.

1) The communication with stakeholders was an obvious topic to look for in the given project because different people with very diverse background worked together (different nationalities, languages, educational backgrounds, development approaches, work experience). Moreover, as already mentioned, the project was not in line with the main business of the organizational unit in which it took place. This resulted in uncertainties who the stakeholders were and how the communication with them was influenced by being out of their everyday business.
2) Requirements engineering was the second evident topic to investigate in the study because the emergence of new requirements partially triggered this redevelopment project. Furthermore, the participants of the project were mainly long-term CERN employees who had a very "technical" and "implementation oriented" view about requirements. Finally, the project was characterized by having no "real clients". This suggested requirements issues concerning "who" would be the stakeholders.
3) At last, the software development process and its distinct phases were interesting topics. This was largely motivated by the specific context of the project: a nonprofit research organization with no time-to-market pressure, but with a project deadline to meet and limited resources. Moreover, the project participants had different software development approaches and varying levels of experience. In addition, the fact that the project had different priorities for the different project participants was suspected to have an influence on its progression and thus on the development process.

As described in Section 2.3.2, as much data as possible was collected during the course of the project. From the beginning, the data collection was targeted towards the initial research topics. Since data collection was as extensive as possible within the given limits of the project, the exact formulation of the research questions was not necessary to drive the data collection. It was postponed until after the operational end of the project since it would not have had a strong influence on the collected data. The research questions of this case study were therefore elaborated based on the initial research topics. This was done after data collection was finished, but still before data analysis could begin.

The first step after data collection was to review the data and elaborate a coding scheme that suited the initially defined research topics. In order to be as relevant and unambiguous as possible, the coding scheme was specified together with the detailed formulation of the final research questions.

The initial research topics defined above were given by the very specific context of the project. During the course of the project, it turned out that issues of communication and domain

knowledge were predominant and strongly influenced the software development process as a whole and especially the activities related to requirements. This guided the initial research topics closer towards these issues. The procedure of data coding and analysis then indicated that also the communication issues were related to issues concerning the exchange of domain-specific knowledge. The key concept is therefore the *domain knowledge* (see 2.2.3.2) as it links the different issues observed during the project. Hence, the final research questions were formulated specifically around issues relating to domain knowledge and their effect. The research questions of this case study can thus be formulated as follows:

1) How can issues related to domain knowledge influence the communication between the participants of a software development project?
2) How can issues related to domain knowledge influence the development process in general and requirements related activities in particular?
3) How can issues related to domain knowledge be counteracted?

2.2.3 Research Context

This section is dedicated to the context surrounding the research questions posed for this study. A literature research phase was conducted after the operational end of the project together with the coding of the collected data. This overview of literature sources is divided into three parts, which correspond to the three main elements of the research questions of this study: communication issues in software development projects, domain knowledge and related issues, and the software development processes. Considering the large amount of existing literature on these three topics, only a limited overview can be presented here. Sources have been selected to concentrate on the aspects relevant to this study.

2.2.3.1 Communication

For clarification, it should be noted first that the term *communication* in the context of this study does not refer to the technical meaning of the term (i.e. data transmission), but it refers to the exchange of information (i.e. verbal interaction) between people. When the technical meaning of the term is meant, it is made explicit by the context (notably in 3.1.2).

Human interaction is well known to be an essential factor in software development activities [4 p. 65]: “Effective communication among the stakeholders of a software development project is crucial to its success.” This fact is far from being new. It has been widely acknowledged in the software engineering discipline and is therefore an often-recurring topic [64].

In one of the most widely known books about software engineering, “The mythical man-month” [22], Frederick Brooks stated in 1975, based on his experience with the development of the IBM OS/360 system, that communication is an essential factor in software development [22 pp. 74-75]. Later, in 1987, he reaffirmed in his famous paper “No silver bullet”: “From the complexity [of software] comes the difficulty of communication among team members, which leads to product flaws, cost overruns, schedule delays.” [21 p. 11, 22 p. 183]. What Brooks

described here is not that communication issues are the source of software development problems, but that they originate from the complexity of software itself. He recognized this inherent complexity to be one of software's "*essential properties"* [22 p. 183]. Brooks identifies the origin of the complexity of software as:

> "I believe the hard part of building software to be the specification, design, and testing of this conceptual construct [software entities], not the labor of representing it and testing the fidelity of the representation." [22 p. 182]

This indicates that the difficulties of communication arise from the lack of a common vision of the conceptual construct that any software entity naturally is. In other words, as described by [64, 109], a *common vision* shared among all stakeholders of a software development project is necessary in order to succeed. Such a common vision is supposed to be a "natural" outcome of any software project in the ideal case: All stakeholders involved need to communicate in order to establish a shared common understanding of the software product that has to be built based on the requirements.

This link between communication and requirements has also been studied extensively. Some papers that were found to be interesting for this study are [55, 2]. They suggest that there is a relation between communication issues and issues with the software development process and requirements engineering in particular. Moreover, one notable study was conducted as an empirical analysis of requirements engineering problems in twelve software companies [53]. It reports that the "developer communication" is the most likely organizational-based issue being at the origin of requirements problems [53 p. 155]. Another empirical study conducted by Kraut and Streeter investigated coordination techniques in 65 software development projects [64]. Among other findings, they conclude that especially *informal communication* should be supported next to the necessary *formal communication* in large projects.

A possible solution to deal with communication issues in requirements engineering is presented in [38]: A framework for avoiding and analyzing communication issues during requirements engineering is explained in detail and its application is shown in comparison with various requirements elicitation methodologies. Complementary to this framework, the same authors conducted an empirical study to support their findings [37].

Al-Rawas and Easterbrook conducted a field study explicitly on communication problems in requirements engineering using interviews and surveys [2]. They concluded that "[…] organisational and social issues have great influence on the effectiveness of communication." [2]. A similar statement was found in [4 p. 65]: "Effective communication is critical to the success of a software development project. […] Yet communication is generally left unsupported by the software development process and by the communication infrastructure." The authors suggest a conceptual framework called "design intent" as a possible solution.

Besides the sources presented above that deal more or less directly with communication issues in software development projects, relevant literature also exists that addresses interpersonal communication issues in general.

A notable study of this type is [81], which investigates the relationship between communication frequency and team performance in project teams. The finding is that too few as well as too much communication can negatively affect the performance of a team. Unfortunately, the employed scale in the presentation of the findings does not allow any conclusions on how much communication really is too few or too much.

In contrast, two other interesting studies of a similar type investigate how communication in a team takes place during design activities [78, 105]. Both analyze the different stages through which a team goes while elaborating a design to solve a given problem.

The variety of the cited papers shows the different points of view concerning human communication issues in development projects. They show that communication is an essential part of any engineering effort in which more than a single person is involved. This demonstrates the importance of this research topic. In the scope of the present work, issues with communication that occurred during the project under study were analyzed and interpreted. This analysis showed that they were related to issues concerning the transfer of domain knowledge.

2.2.3.2 Domain Knowledge

Before presenting relevant literature about domain knowledge and its link to communication, an overview shall be given how domain knowledge is defined according to the various sources.

One of the most highly cited studies dealing with domain knowledge in software engineering [42], conducted by Curtis, Krasner, and Iscoe, does not explicitly define the term but introduces it as the "[…] deep application-specific knowledge required to successfully build most large, complex systems […]" [42 p. 1271]. The second widely cited paper in this context [109] takes over the concept from [42] and adds:

> "Knowledge is the raw material of software design teams. […] In general, individual team members do not have all of the knowledge required for the project and must acquire additional information before accomplishing productive work. […] Productive design activities need to revolve around the integration of the various knowledge domains. This integration leads to shared models of the problem under consideration and potential solutions." [109 p. 63]

The two influential papers both refer to the publication of Adelson and Soloway from 1985 [1], which was one of the first representative empirical studies investigating the role of domain knowledge in software engineering projects. Although Adelson and Soloway use the term "domain experience" instead of "domain knowledge", the underlying concept is the same:

> "A designer's expertise rests on the knowledge and skills which develop with experience in a domain. As a result, when a designer is designing an object in an

> unfamiliar domain he will not have the same knowledge and skills available to him as when he is designing an object in a familiar domain." [1 p. 1351]

These descriptions of the term domain knowledge do not explain what type of knowledge is meant. One paper was found to discuss what type of knowledge is relevant in software engineering [91]. The author differentiates between *topic knowledge* and *episodic knowledge*, which are both required in application development:

> "Topic knowledge refers to the meaning of words, such as definitions in dictionaries and textbooks. Topic memory is made up of all the cultural structures of an environment and supports the organization of knowledge related to an environment. […] Episodic knowledge consists of one's experience with knowledge. Examples include reusing a function, decomposing data-flow diagrams, defining objects from specification requirements, building entity-relation graphs, and documenting programs. Most of these activities are learned through experience once the topic knowledge is obtained from textbooks or courses." [92 p. 88]

According to [92], the lack of episodic knowledge about the application domain can lead to "[…] well-designed but inappropriate software […]" and "At the coding level, lack of episodic knowledge in the programming language sometimes results in an unduly complex program." [91 p. 88].

Another differentiation of knowledge types is given by [107]. Based on [97] but without going into details, the study differentiates between *technical knowledge* and *application domain knowledge*:

> "Technical knowledge refers to knowledge about design (e.g. design patterns, heuristics, best practices, technical constraints, and estimation models), programming (e.g. programming languages and development tools), and software processes (e.g. methodology, code testing and debugging procedures). Business application domain knowledge refers to knowledge about the customer's business processes, business rules, activities, stakeholder needs, and the customer's business objectives for the software." [107 p. 900]

The problem with this definition of domain knowledge as "*business application domain knowledge"* is that it might well be applied to general-purpose application software, but it does not take into account the additional knowledge necessary for domains with specialized interfaces to other software or hardware systems (especially interfaces to legacy systems). For such kind of domains, an essential part of the domain knowledge consists of the knowledge about the surrounding technical infrastructure like specialized libraries, specific hardware capabilities, network interface, etc. However, such technical knowledge is specific of the domain and is therefore different from the "general" technical knowledge defined by [107]. As [97 p. 27] puts it: "Software development requires access to knowledge not only about its domain and new technologies but also about the domain for which software is being developed." Thus, it is useful to extend the differentiation of [107] by adding this additional

type of knowledge under the term *technical domain knowledge*. To sum up, the knowledge necessary to develop software can be divided into three types:

- *General technical knowledge*, which is general knowledge about programming languages, software development processes, design patterns, etc.
- *Technical domain knowledge*, which is the knowledge about the specific software and hardware infrastructure of the environment for which the software is built.
- *Business domain knowledge*, which is the knowledge about business processes, procedures, rules, and user needs of the domain for which the software is built.

Thus, in order to define the term *domain knowledge*, it is possible to split it up into *technical domain knowledge* and *business domain knowledge*. This twofold definition is well suited for the analysis of the domain knowledge issues that occurred within the project under study.

Based on the definition of domain knowledge, it is possible to proceed on defining the concept of *domain knowledge integration* in the context of software development. In every software development effort surpassing a certain basic level of complexity, it is virtually impossible for any single individual to perform the complete development on his own. It can be "[…] beyond the understanding of any single person." [106 p. 15]. Therefore, the focus lies on software development in teams:

> "In general, individual team members do not have all of the knowledge required for the project and must acquire additional information before accomplishing productive work. […] Productive design activities need to revolve around the integration of the various knowledge domains." [109 p. 63].

According to this definition, domain knowledge integration means that every team member acquires the level of knowledge he needs for a given project so that the team as a whole is able to understand the requirements and subsequently elaborate and implement a proper design.

A slightly different approach is described in [107 p. 900]:

> "We define this process of combining dispersed business domain and technical knowledge and embodying it in the design as knowledge integration."

Both definitions seem appropriate. A developer of software in a given domain needs to integrate all different types of knowledge and thus perform *domain knowledge integration* to fully comprehend the customer's requirements. This is best illustrated by citing an engineer interviewed for the study presented in [42]: "Writing code isn't the problem, understanding the problem is the problem." [42 p. 1271]. Furthermore, "Many forms of information had to be integrated to understand an application domain." [42 p. 1271].

What is missing in the definitions is the explicit differentiation of the types of knowledge that have to be "integrated". By merging this aspect into the definition given by [107], *domain knowledge integration* is to be defined as *the process of acquisition of domain knowledge (both*

types) in order to comprehend the requirements and to enable their subsequent condensation into an appropriately designed and implemented solution satisfying the customer's needs.

Having defined the terms *domain knowledge* and *domain knowledge integration* within the context of this study, an overview of the relevant findings from the literature about this research topic is given. To begin the literature overview, it is interesting to note that this research topic is not new and is recurring even in recent publications. The importance of domain knowledge in software engineering is known since the 1980s as shown by the two influential studies [42] and [1]. Recent studies, like [107] from 2004, show that the influence of domain knowledge on software development projects is still a research topic in the software engineering discipline:

> "Software development is a knowledge intensive process that involves the coordinated application of a variety of specialized knowledge in conceptualizing and designing a coherent software solution for a business problem." [107 p. 899]

One of the first empirical studies conducted about domain knowledge in software engineering is [1] in which three expert and two novice software designers were observed while performing predefined software design tasks. The designers were asked to elaborate designs in a domain in which they had previous experience and in domains in which they had no experience. The study identified several handicaps that were observable only when a designer did not have the necessary domain knowledge for the given task. This insight opened the way for more subsequent empirical studies.

The probably most influential study dealing with domain knowledge in software engineering is [42] in which structured interviews with developers and project managers from 19 industrial software projects were conducted and analyzed. In this study, three main issues of software development efforts were identified:

> " (1) the thin spread of application domain knowledge
> (2) fluctuating and conflicting requirements
> (3) communication and coordination breakdowns " [42 p. 1270]

Therefore, [42] is relevant in all points to the research topics of the present study. Concerning the domain knowledge problem, the findings of [42] showed that:

> "This problem was especially characteristic of projects where software was embedded in a larger system (e.g., avionics or telephony), or where the software implemented a specific application function (e.g., transaction processing)." [42 p. 1271]

Moreover, in [42], domain knowledge deficiencies were found to be responsible for specification mistakes because of misinterpreted requirements, to have an impact on team interaction, and to introduce apparent requirements fluctuations. This view is also supported by [38 p. 52]: "Designers tend to lack domain or business knowledge and consequently tend to misunderstand or ignore some requirements and their social context."

A different approach to the topic of domain knowledge was taken by [109], by investigating the acquisition, the sharing, and the integration of domain knowledge within an industrial software development project. In this study, several meetings of a software project team were videotaped and analyzed. The meetings ranged from the project's startup until its shift to the final coding phase. The published findings show that discussions about domain knowledge were one of the three major things observed in the videotaped meetings [109 pp. 67,69]. The observed team had to go through a costly learning curve in order to acquire the missing domain knowledge [109 p. 74]; therefore, the addition of domain experts to an existing software development team is recommended by the authors [109 p. 75].

The quite different empirical study presented in [107] was conducted as a survey on 232 software projects. It is described as being the first large scale empirical study about the influence of knowledge integration on the performance of software development [107 p. 899]. The study showed that a higher degree of knowledge integration led to higher effectiveness, lower cost overruns, and a lower defect rate of the final software product [107 p. 904].

Another paper relevant to the research topic is not a report about an empirical study as the previous references, but it presents a tool that is aimed at managing design related knowledge [106]. What is interesting is that the paper acknowledges the importance of knowledge in software development and defines "community specific folklore" to be the essential part of knowledge necessary for successful development [106 p. 15]:

> "Much of the knowledge required to be a successful developer in a particular organization is community specific, concerning the application domain, the existing software base, and local programming conventions. This knowledge typically has the status of folklore, in that it is informally maintained and disseminated by experienced developers."

This view complements well the broader definition of domain knowledge adopted in the present study. Moreover, it indicates a strong link between domain knowledge integration (the dissemination of the folklore) and communication as it is transferred directly from experienced developers to less experienced ones. This strong link between knowledge transfer and communication is also supported by [97 p. 28]: "Communication in software engineering is often related to knowledge transfer."

As a solution to improve this inefficient and fragile way of domain knowledge transfer (integration), [106] suggests the use of a database tool to maintain and disseminate the knowledge. It is interesting to note that other sources like [45] are opposed to this solution and suggest on the contrary the focus on communication: "It is impossible to capture all expertise in databases. Hence technology must move away from this goal and foster communication." [45 p. 101].

By turning the attention towards the research question about the influence of domain knowledge on the software development process, it is possible to find indications about such a relationship in the literature. To start with, [86], which is a discussion about research in

general in the software engineering discipline, indicates a strong relationship between domain knowledge and requirements engineering: "Lack of domain knowledge is one of the most significant impediments to successful requirements capture." [86 p. 23]. In addition, the study presented in [107] led to the conclusion that in the case of complex domain knowledge (knowledge about customer needs) "[…] formal requirements are likely to capture only a subset of a project's true requirements." [107 p. 905]. This shows that requirements engineering can be hindered in environments with complex domain knowledge when this issue is not explicitly tackled. Moreover, one of the findings of the already presented study [42 p. 1275] is that requirements are apparently "fluctuating" when essential domain knowledge is missing in a development team, thus impeding a proper requirements engineering phase on the beginning of a development process. Furthermore, the same study [42 p. 1284] suggests that the "[…] developers held a model of how software development should occur, and they were frustrated that the conditions surrounding their project would not let them work from the model." The "conditions" mentioned here were among others the lack of domain knowledge [42 p. 1270], and this reinforces that there is a strong influence of domain knowledge on the development process within a project.

Not only the requirements engineering phase of a software development effort seems to be influenced by domain knowledge issues. The study of [109 p. 67] revealed that although a sequential process was intended in an observed project, the phases of the project were not sequential and independent of each other: Acquisition of domain knowledge interfered with the evolution of the different phases indicating an influence on the software engineering process as a whole. This is furthermore supported as the results of [91] indicate that software designers take a more or less systematic approach in developing software depending on their confidence in having "[…] access to all the knowledge required to do the task." This suggests that systematic software development is only possible if all necessary domain knowledge has been successfully integrated.

2.2.3.3 Software Development Process

The strong link between domain knowledge and the software development process indicated in the literature opens the question of what exactly is meant by the term "software development process". What is to be understood by this general term is not always consistent in literature. Since it is one of the main topics of the present study, the differing notions are overviewed before presenting relevant contributions from the selected literature.

A general explanation of what a process in engineering is was found in [39 p. 101]:

> "The ultimate goal of the theories, techniques, and methods that underlie any engineering field is to help engineers in the production of quality products in an economic and timely fashion. […] The product is *what* is visible to the customers, and thus it is all that matters in the end. The process is *how* this goal can be achieved."

Similar to this, but more concrete, is the definition given in [47 p. 126]:

> "*Process*: The (partially ordered) sequence of activities, entrance and exit criteria for each activity, which work product is produced in each activity, and what kind of people should do the work."

In the field of software engineering, two slightly different notions can be differentiated: on one hand the "software process" and on the other hand the "software engineering process". *Software engineering* itself can be defined as:

> "Software engineering refers to the disciplined application of engineering, scientific, and mathematical principles and methods to the economical production of quality software." [58 p. 82]

Based on this meaning of the term "software engineering", a *software engineering process* can be defined as:

> "The software engineering process is the total set of software engineering activities needed to transform a user's requirements into software." [58 p. 83]

Accordingly, the software engineering process is the process pursued when developing software. It can thus be called the *software development process* to distinguish it from the broader term *software process*, which encompasses more than just the development or engineering activities to produce software:

> "*A software process can be defined as the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain a software product.*" [50 p. 28]

The definition of the term "software processes" is closely related to the concept of *software lifecycle,* which can be defined as:

> "In general, a software lifecycle defines the *skeleton* and *philosophy* according to which the software process has to be carried out. However, it does not prescribe a precise course of actions, an organization, tools and operating procedures, development policies and constraints." [50 p. 28]

Based on these definitions, an overview of selected contributions to the field of software processes is given.

A good starting point for software process research is [50], which presents along with the concise definitions a historic overview, a critical evaluation, and an outlook for this research domain. This publication makes the fundamental motivation of process-related research evident, which is the "[…] direct correlation between the quality of the process and the quality of the developed software." [50 p. 27]. Moreover, it confirms that "[…] *developing software is a complex process* […]" and that "[…] the quality of a software product heavily depends on the people, organization, and procedures used to create and deliver it." [50 p. 28].

Another overview of software process research in the past decades and an outlook to the future can be found in [39]. The paper covers two different aspects: In its first half it describes how to look at software processes either as black boxes or as transparent processes and treats the different historic approaches to solve the inherent issues of software processes. The second half of the paper focuses more on the field of process programming and process-centered software engineering environments. Nevertheless, the paper is a useful source in order to understand the evolution of software process research. It is especially interesting that, according to [39], the informal nature of initial product requirements and their "inevitable tendency" to change over time are the inherent problems that software processes have to deal with. The basic principle to overcome these difficulties is to give software developers continuous feedback during the development process [39]: "This may reduce the time between making a decision and discovering that the decision was actually wrong, thus reducing the costs needed to develop an acceptable product." To implement this solution, the paper suggests incorporating the customer into the process by letting him observe various artifacts created during the development and thus making him understand its evolution. Based on these basic principles, the paper gives then a critical analysis of past tentative solutions to software development problems like the sequential software lifecycle, formal and informal methodologies, and automated development attempts.

Boehm's influential paper from 1988 defining the *spiral model of software development and enhancement* [15] shows that he founded the development of his spiral model on known issues with previous methodologies. That paper thus begins by giving an excellent overview of the evolution of different "process models" [15 pp. 61-64]. Boehm differentiates between a "software process model", which is aimed at determining "[…] the *order of the stages* involved in software development and evolution and to establish the *transition criteria* […]", and a "software method", which determines "[…] how to navigate through each phase […] and how to represent phase products […]" [15 p. 61]. Looking back at the definition of a "software process" adopted within the present study (see above), it roughly corresponds to a combination of both of Boehm's concepts: a "process model" together with one or several "software methods". The "process model" alone is comparable to the notion of "software lifecycle" as defined in [50] (see above) but at the same time it should be noted that it differs from Boehm's "life-cycle concept".

The historic evolution of software engineering traced by the paper [15] starts with the "code-n-fix" strategy and proceeds with "stagewise and waterfall models"[1], the "evolutionary development model", and the "transform model" and it points out each model's difficulties. Common problems of those process models are the risk that the final product is not satisfying user requirements and/or issues with the product's quality and efficiency. As a solution, Boehm suggests the "spiral model" which is a cyclic process approach tightly bound to project risk management. It is intended to be flexible enough to "[…] accommodate most previous models as special cases […]" [15 p. 64].

[1] See [96] for the formal description of what is called the "waterfall model".

Another paper by Boehm [17] tries to explain that the inability of creating a complete specification before starting a software development makes a strictly sequential lifecycle approach and evolutionary development problematic. The paper notes that the difficulties lead to the proliferation of a variety of processes, which then make the selection of one specific process difficult. Boehm notes: "In many cases, organizations have remained loyal to admittedly flawed models - such as the waterfall - because they believe that the value of any common framework is worth the price of its imperfections." [17 p. 75]. As a solution, Boehm suggests three essential milestones that any process should manage in order to be successful. He elaborates his new approach in the paper [17], but unfortunately, it seems to be targeted only towards very large development projects. It is interesting to mention that Boehm based this theory on many analyzed successful and unsuccessful implementations of his famous spiral process.

The most revealing and complete overview of software processes was found in [16], where Boehm traces, according to his view, the history and evolution of software engineering practices. He proposes a grouping of common practices by decades and evaluates them by summarizing lessons learned from each of those decades. The lessons learned with the highest relevance for the research topics of the present study are [16 pp. 24-25]:

- "Avoid using a rigorous sequential process"
- "Eliminate errors early"
- "Determine the system's purpose. Without a clear shared vision, you're likely to get chaos and disappointment."
- "Avoid Top-down development and reductionism."
- "What's good for products is good for process, including architecture, reusability, composability, and adaptability."
- "Consider and satisfice all of the stakeholders' value propositions."

In addition to this analysis of software engineering practices, Boehm [16] offers an extensive collection of literature resources, including several highly influential works of the software engineering domain.

A notable paper in the context of this study is "A rational design process: how and why to fake it" from Parnas and Clements [80], which was also published in [79]. The paper shows that already in 1986 the software community was aware of the fact that strictly sequential processes like the "waterfall" stemming from the 1970s were not applicable to most real software development projects. Nevertheless, sequential processes are still being adopted nowadays for software projects [39].

The following reasons might explain why knowingly flawed sequential approaches such as the "waterfall" are still in use:

- The proliferation of diverse advanced software processes lead companies to stick with sequential processes because they believe that the benefit of having a common process that is understood by everyone is worth its flaws [17 p. 75].

- A sequential approach naturally delivers documentation artifacts, which are a recognized standard in industry [80, 15 p. 63].

The solution suggested by Parnas in 1986 [80] is to keep the apparent structure of a sequential waterfall approach in the documentation, but to work in reality in a more flexible way, using any other process if necessary. This solution keeps the advantages of having a "common standard" and a comprehensive documentation and at the same time deliberates the developers from the strict sequential approach. This is what Parnas called "faking" the process.

To sum up, there is probably no stronger statement about strict sequential approaches than:

> "No system has ever been developed in that way, and probably none ever will. Even the small program developments shown in textbooks and papers are unreal. They have been revised and polished until the author has shown us what he wishes he had done, not what actually did happen." [80, 79 pp. 356-57]

Finally, a paper from Larman and Basili [68] was found to explain the controversy around sequential "waterfall"-style development approaches in the historical context surrounding iterative incremental development processes. In this paper, Larman and Basili show that the "waterfall" process was initially not intended to be a strictly sequential approach. Nevertheless, this misinterpretation became widely adopted for software development projects since the 1970s, apparently because of its "simplicity":

> "Software development is a very young field, and it is thus no surprise that the simplified single-pass and document-driven waterfall model of "requirements, design, implementation" held sway during the first attempts to create the ideal development process." [68 p. 55]

The paper suggests that iterative and incremental development (IID) is the origin of the agile processes, which are applications of IID [68]. They promote the success of IID by citing several success stories and by combining the views from different influential sources. Among others, they also refer to the works of Parnas, Brooks, and Curits cited above. As an example of a success story, they cite the case of a large air traffic control software project that became a success by switching from an initial waterfall approach to an iterative incremental development [65]. Moreover, they refer to a paper from Basili and Turner [7] as one of the early contributions to IID. In this paper [7] the authors state:

> "Building a system using a well-modularized, top-down approach requires that the problem and its solution be well understood. Even if the implementers have previously undertaken a similar project, it is still difficult to achieve a good design for a new system on the first try." [7 p. 390].

This acknowledges that when domain knowledge is missing, a strictly sequential development approach is not realistic[2]. As a solution, IID allows "[…] to develop a software system incrementally, allowing the developer to take advantage of what was being learned during the development of earlier, incremental, deliverable versions of the system. Learning comes from both the development and use of the system, where possible." [68 p. 49]. Iterative incremental development approaches thus correlate well with the recommendations for software development processes of Boehm [16] cited previously.

The selected literature presented above shows the diversity and controversy that exists about the topic of processes in the software domain.

2.3 Case Study Design

Essential parts of an empirical study report are the description of its design, the detailing of the approaches taken by the researcher, and the methods employed for the collection and analysis of the empirical evidence. The importance of such detailed descriptions is imposed by the necessity to make the study credible and replicable. Especially in case studies, the exact procedures undertaken by the researcher need to be comprehensible and repeatable [111 p. 38]. The following subsections are therefore intended to report the approaches and methods employed within this study. First, the overall methodology of the study is described, followed by an explanation of the process of data collection and data analysis. Finally, possible threats to the validity and the quality of the study are addressed.

2.3.1 Method of the Study

Different types of empirical methods exist (see 2.1.1). Based on the constellation of the given project, only two types of methodologies were found to be suited depending on their characteristics as presented by Yin [111 pp. 5-7]. These were the *case study method* [111, 103, 110] and the *history* that is comparable to what is called a *post-mortem analysis* in the literature [110, 34, 104, 13].

In order to differentiate the available empirical methods, Yin [111] cites three criteria or conditions:

> "The three conditions consist of (a) the type of research questions posed, (b) the extent of control an investigator has over actual behavioral events, and (c) the degree of focus on contemporary as opposed to historical events." [111 p. 5]

As defined in Section 2.2.2, the research questions posed are questions of the "How?" type, which eliminates *surveys* and *archival analysis* as research strategies according to Yin [111 p.

[2] At this place, it should be pointed out that the retrospective analysis of the project under study showed (see Chapter 4) that also in the given case a strictly sequential software development process following the waterfall model did not succeed. Such an approach was intended initially. The project then switched to an iterative incremental approach, informally at first, and then in form of the agile Scrum process.

5]. Furthermore, as explained in Section 2.2.1, the setting and the characteristics of the project under study were predetermined and could be influenced only marginally for the purpose of the study. This necessitated a method requiring only limited or no control over the study object [111 pp. 5-9]. Finally, the focus during data collection was on contemporary events of the evolving project rather than on historical events predating the project. According to Yin [111], all these characteristics match the required conditions for employing the case study method.

The constellation of the project and its study necessitated that large parts of the study-related activities, namely data coding, analysis and interpretation, were executed only with a notable delay after the operational end of the project. This was caused by the fact that the author was at the same time actively involved in the project as the main fulltime software developer (see 2.2.1). Therefore, the elaboration of the case study in parallel with the active project participation was not possible. One could argue that because of this "retrospective" aspect, the empirical method suitable would be the *history* or *post-mortem analysis*. Nevertheless, the overall characteristics of the project and the study conditions favor the case study method. This is especially emphasized as a case study "[…] investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident." [111 p. 13]. Because of the predominant role of the project's context for the research questions (see 2.2.2), the case study was found to be the best suited method.

The retrospective aspect evoked above was ignored since it was found to have little importance for the study methodology. Yin notes: "Even though each strategy has its distinctive characteristics, there are large overlaps among them." [111 p. 5]. In addition, he confirms that especially "[…] case studies and histories can overlap […]" [111 p. 8].

Within the chosen case study strategy, there are several types of possible approaches. In general, case studies can be explanatory, descriptive or exploratory [111 pp. 4-8, 112]. Considering the type of the research question and the context (see 2.2), this study can best be characterized as being descriptive and explanatory. It is a descriptive study because one of its goals is to describe the evolution and outcome of the project conducted within its very specific context. The study aims at showing "what was done" using the coherent set of methods of case study analysis and thus avoiding the pitfalls of producing only a plain elicitation of facts [82 p. 349]. At the same time, the study has an explanatory character by investigating the posed research questions using the case study methodology. By doing this, the explanatory part strongly relies on the descriptive part since a detailed description of the object under study and its context is necessary for the outcome of the case study to be credible and significant [57 p. 481, 60 p. 723, 82 p. 351].

This twofold approach is also reflected by the outline of this case study report: Chapter 3 is dedicated to the description of the project and its context, whereas Chapters 4 and 5 represent the explanatory part of the study.

2.3.2 Data Collection

One of the most essential steps in conducting a case study is the collection of evidence about the object under study from a variety of sources [111 p. 83]. The specificities of case studies in the field of software engineering, already introduced in Section 2.1, and the specific context of this study, described in Section 2.2, influenced the available sources and possibilities of data collection. The data collection approach employed for the present study is described in the following.

The present case study investigates a real project in its real-life environment. Therefore, it can be characterized as an "in vivo" study [108]. This is exactly the kind of studies that researchers are urging to perform since they allow investigating the human factors of software development. Unfortunately, for this specific kind of study, there is no explicit guide on how to perform proper data collection. Therefore, the general principles about the collection of case study evidence applying to other research fields were taken as the basis for the present case. A good overview of these methods can be found in [111 pp. 83-108].

The employed data collection methods had the initial goal to collect as much qualitative data as possible. The constraints were such that the collection methods should not affect productivity during the project and be as little intrusive as possible. As explained in Section 2.2.1, the author of this study was on the same time the main software developer during the project. Moreover, he was the only project participant working on a fulltime basis on the project during its whole timeframe (11 months). Thus, his activities in his role as main developer and all his interactions with other participants were in the primary focus of data collection. A very appropriate data collection method for this kind of constellation was found to be personal note taking, executed by the main developer.

The process of personal note taking can be seen as a special case of the "participant observation" method described in [111 pp. 93-96]. In this case, the observer and the participant being observed are the same person. The form of note taking based on observation is also called *field notes* in [99 p. 559]. The classic participant observation by an external observer, especially of software engineers, has several flaws:

- The observer only observes *external activities* of the developer. The essential part of intellectual work done during software development takes place in the head and stays hidden for any external observer [99 p. 558]. A possible solution is to perform so-called "think aloud protocols" for which the observed software developer has to speak freely about what he is currently thinking during his work [99 p. 558, 46].
- Detailed observation can be intrusive and thus jeopardize the "in vivo" conditions of a case study. "The participant observer must take measures to ensure that those being observed are not constantly thinking about being observed." [99 p. 558]. Methods that try to capture the hidden intellectual part of software development activities, like the "think aloud protocols", are even more intrusive than classical observation and cannot be conducted during the whole timeframe of a real-life software development project.

Therefore, the alternative method of personal note taking, which has been employed, can be a viable workaround to avoid the issues above. To contrast it with the classical "external" observation, it can be called "internal participant observation". Nevertheless, the "internal" observation has also drawbacks. The critical aspects of this data collection method are:

- The effects of the personal bias of the note taker have to be taken into account. These can be positive and negative effects. On the negative side is that the personal beliefs and the subjective filtering during note taking corrupt the objectivity of the data. If there had been an additional external observer participating in the study, any possible bias would certainly have been largely reduced. Unfortunately, this was not possible within the given constraints of this project. On the positive side is that it allows analyzing the case from the point of view of a specific participant, which can potentially bring new insights for the case study.
- The level of detail in the note taking process has to be appropriately chosen. Too detailed notes take too much time and reduce the participant's productivity. Too few notes can lead to missing essential information for the subsequent analysis of the data.
- The process of taking the notes needs to be as little intrusive as possible. The participant has to be free in order to work in almost "normal conditions", and he shall not be forced to concentrate too much on the note taking activity.

However, there are several possibilities to control these critical aspects of the method. The following concrete measures were taken during the data collection for the present study:

- Imposing a specific way or framework for the note taking activity can limit the impact of the personal bias. To counteract the personal bias, notes during the study were taken with two levels of focus: The first level consisted of noting all current activities and external facts as objectively as possible. These were mainly notes aiming at answering the following question: "Who did what when?" This is similar to the "time log" method suggested by [99 p. 559]. On the second level, private notes were taken capturing subjective impressions of the note taker about his personal work and external interactions and influences. By explicitly being free to write down those impressions, the temptation of letting them influence the information of the first level was expected to be reduced.
- To master the degree of detail needed for the notes, the guidance imposed was to put the maximum possible level of detail on the first level of notes: Events and interactions with other individuals during the project were reported with as many details as possible. The degree of detail of the second level of notes was deliberately free. Therefore, the total amount of information collected over time varied.
- In order to stay as little intrusive as possible, the note taking was performed in several steps:
  - Unexpected events were immediately noted as they happened and were refined with additional details at the end of every workday of the project if necessary.

- Activities, along with their durations, were noted at the transition times between one activity and another or whenever not possible, they were noted retrospectively at the end of every workday.
- Meetings with other participants were always accompanied by note taking. Additional details were then added after the end of each meeting.

For clarification, it should be mentioned that the first level of note taking always included the name of the participating persons in an activity or interaction and the time of occurrence together with its duration, when applicable. This quantitative information constitutes an essential basis for the analysis of interaction and communication issues that was conducted during data analysis.

The following describes how the different data sources were collected by the main developer on an everyday basis during the course of the project.

The workplace of the main developer was equipped with two computer screens. On one screen, there was always a digital note-taking application running in the background to be at immediate disposition when needed. The notes in this application were taken in the form of diary notes in chronological order. The structure was maintained using timestamps and special reference keywords to identify explicitly reported meetings and interactions. Especially important were the timestamps that served later on as a basis for a retrospective quantitative analysis. A comparable approach, although in a different constellation, was found in [84], where a "time-diary experiment" was conducted with 17 software developers during a timeframe of one year.

In addition to these diary notes taken directly in digital form, handwritten notes were taken during meetings. Separately, meetings were also reported within the digital diary notes. The handwritten meeting protocols were time-stamped and collected in chronological order. Those collected handwritten notes then served during analysis as a source to crosscheck the correctness of the data derived from the digital diary notes. Thus, a second source of data was available as collected case study evidence. Moreover, as another separate data source, detailed handwritten records about the main developer's personal work time were kept. These allowed a crosschecking of timestamps and served as basis for the evaluation of the data-gap as explained below.

In order to collect as much relevant evidence as possible during the case study, another completely independent source of data was used. Since the purpose of the project under study was to develop a software package, an important amount of software source code was produced during the project. This source code was regularly committed to a concurrent versioning system (CVS) that was available on site. Committing to the CVS server was a necessary step since it was required for compilation and testing of the software (see Section 3.1.2.2 for more details). Therefore, every evolution of the software during the course of the project was reflected by the code committed in chronological order to the CVS server. In order to use the data as evidence for the projects evolution over time, a logbook extraction feature of the CVS server was used to acquire a digital chronological list of changes to the developed

software. The information contained in this extracted data was mainly the precise amount of lines of software code added or modified on every entity of the software. A special software tool to analyze and generate statistical information from the extracted CVS logbook was developed by the author for this specific purpose. Using this tool, the evolution of the implementation effort could be traced over time, which represents an additional source of collected evidence. A similar approach is suggested by [82 p. 353].

The last source of data during this study was the collection of artifacts produced as intermediate results during the project such as documents, spreadsheets, screenshots, photographs, software modules and prototypes, and the final software itself. The few documents were mainly technical documentation about relevant interfaces. Furthermore, several important spreadsheet tables were elaborated during the project and were used for requirements engineering and project management purposes. Those artifacts were also collected on a regular basis. Moreover, photographs were taken from the project's laboratory setting and from a whiteboard that was used for project management purposes during meetings in the final phase of the project. Furthermore, screenshots were collected from all the different stages of the software developed within the project. Finally, the new software itself was also included into the collected data. By using a special simulation mode, it is possible to run the software outside of its necessary network infrastructure in order to analyze some of its user interface characteristics and to compare them with the initial requirements.

Some of these types of collected evidence are also suggested by case study literature such as [111], although they are categorized differently. Yin, for example, differentiates between physical artifacts [111 p. 96], documentation [111 pp. 85-88], and archival records [111 pp. 88-89].

The following list is intended to give a full overview of all data sources collected for the purpose of the present study:

- Extensive chronological diary notes (time stamped)
- Meeting notes (handwritten meeting minutes)
- Table of work times (handwritten)
- Logs from the CVS server used for the software development
- Artifacts in the form of:
  - Documents (produced and supplied ones)
  - Spreadsheets (for various purpose)
  - Screenshots (from software applications and prototypes)
  - Source code and executable applications and prototypes
  - Photographs

To complement this overview, it should be mentioned that collecting all electronic mail correspondence between the main developer and the other participants of the project was intended. Unfortunately, due to an unexpected technical problem with the electronic mail exchange server, it was possible to collect only a subset of the communication archive.

Because of being chronologically incomplete, this source of data was discarded. The effect of its loss is considered minor since most communication happened face-to-face among the project participants. Finally, it has to be noted that other possible imaginable sources of data like tape recordings of meetings or additional external observers were not within reach with the given resources and the predefined constellation of the project and its study.

The list of employed data sources above shows that a variety of evidences was collected during the case study. This conforms to the often-recurring recommendation for case studies to use multiple data source whenever possible [111 pp. 97-98]. Moreover, as recommended in [111 pp. 101-05], all the collected data was included in a "database", which consisted of three parts:

- A bound booklet consisting of the printed digital chronological diary notes with a volume of 96 pages.
- A physical office folder with diverse printed or handwritten artifacts (documents, spreadsheets, screenshots, meeting minutes)
- A digital compact disk including all collected digital data as well as the extensive diary notes in digital form for easier analysis.

This case study database is not fully indexed, as suggested in the literature, since it was not found to be helpful during data analysis and because the data was already ordered by its source type and by chronological timestamps. The detailed presentation of the analyzed data in Chapter 4 will show that not all of the collected sources of data were found to be of equal value. The sources that helped to gain more insights than others will be referenced at the appropriate places in Chapter 4.

2.3.3 Data Analysis

In order to describe the methods employed for the case study, this section is dedicated to the detailed description of the data analysis approach and its methods. The data coding and analysis work done for this case study was intentionally postponed until after the operational end of the project and thus after the end of data collection. This was done in order to minimize the influence of the case study on the project itself and to avoid manipulative interactions based on early findings (see also [111 p. 8, 111 pp. 93-96]). The content and the results produced from the data analysis are described in detail in Chapter 4 of this work.

The analysis strategy used for this study was largely influenced by the nature of the collected data. As explained in section 2.3.2, the collected evidence consists of several sources of different types, mostly qualitative in nature. The diary notes, as resulting from the self-observation method, were found to be the most valuable source of data containing an important amount of information about the project with respect to the research topics of this study. In order to make this primary information accessible to a systematic qualitative and quantitative analysis and in order to crosscheck it with other sources, a specific approach was elaborated. This approach was divided into two distinct phases: The first phase was a pre-analysis or pre-processing of the collected evidence in order to make the contained relevant

data accessible for further systematic analysis. The essential method that was used is called *text coding*. The second phase was then concerned with the extraction of relevant qualitative and quantitative facts out of the preprocessed data. The approach served to transform qualitative data into quantitative data [99 p. 563]. The two phases of this approach are explained below.

The first phase of the data analysis approach was based on the following consecutive steps:

1) The first step in the data analysis approach was to perform a thorough reading of the digital diary notes by keeping a focus on information related to the research questions of the study. The goal was to get a first impression of the amount of relevant information contained within the data and to establish the basis for a text-coding scheme [74 pp. 56-57]. The outcome of this first step was a set of *tags* that were employed for the coding. The method of text coding was selected because it was found to be appropriate for the given case. Its benefit was to allow a systematic extraction of qualitative and quantitative facts out of the chronological diary notes.
2) The second step consisted of the actual text coding itself: The previously defined tags, denoting specific occurrences of information in the diary notes, were applied to all corresponding text passages in the diary notes document. The fact that diary notes already contained timestamps and keys facilitated the coding effort and increased its accuracy. For this task, a specialized open-source software package for text coding (for tagging and annotating) named GATE "*a General Architecture for Text Engineering*" [40] was employed. This software package was chosen out of several similar packages that were tested for the purpose of this study because of its user-interface, which was found to be very effective for tagging large amounts of text segments. Furthermore, it offered a very effective export capability for analytical purposes. The GATE software allows the easy overlapping of different tags or annotations over the same segments of text. In order to perform the described tagging of the whole diary notes, the whole document had to be read thoroughly several times, each time focusing on another category of tags. During this on-screen reading within GATE, the tags were subsequently applied in order to produce the final tabular dataset. This dataset being the outcome of this step consisted of a list of references to all occurrences of each processed tag. A screenshot of the GATE tool is shown in Figure 1. The central part of the tool shows the raw text input, i.e. the diary notes with the colored overlays indicating the passages marked with a given tag (participant names were removed for anonymity as explained below). A subset of the tags is shown in the right part of the tool and a subset of the output list of occurrences in the lower part.
3) The following step of the data analysis approach consisted of extracting the tabular data set containing the tags and their occurrences out of the GATE software into a spreadsheet application for further analysis. The obtained spreadsheet was divided into several tables. Each table included the sequential list of all occurrences of a given category of tags. For each tag occurrence, the unique numeric identifier directly linked the entry to the corresponding text passage within the GATE software package.

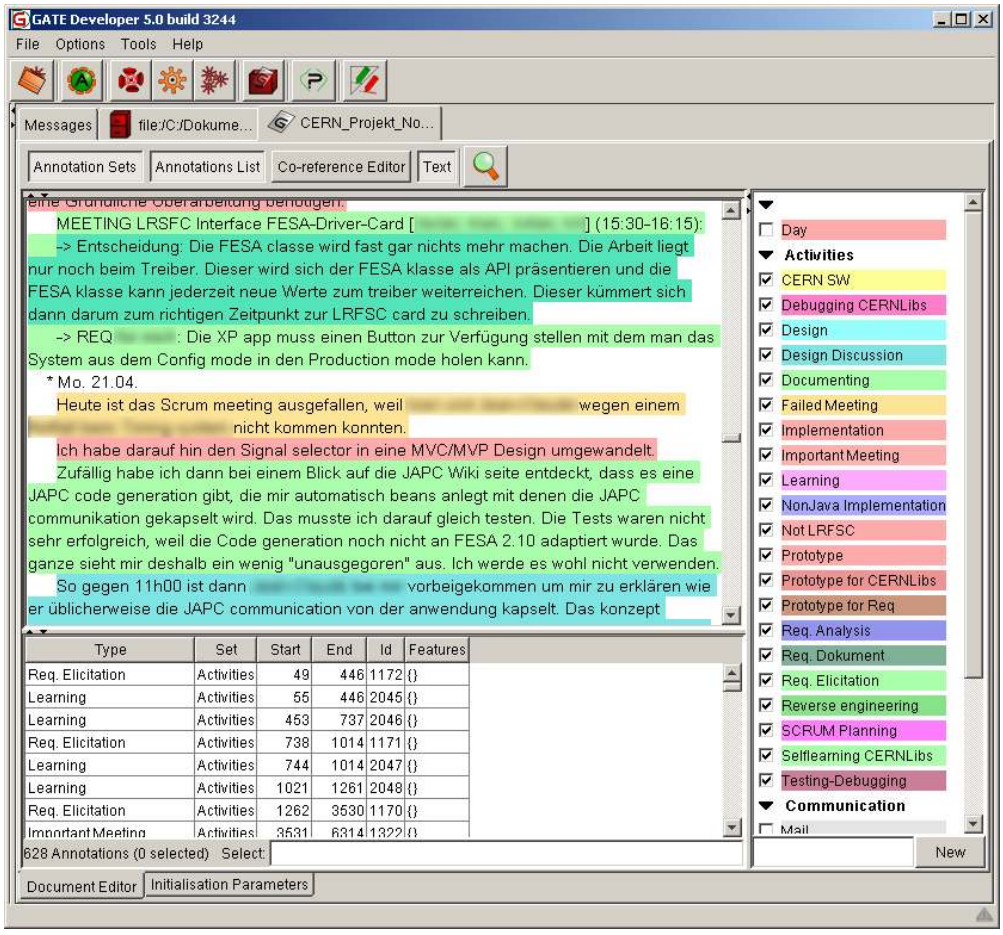

Figure 1. GATE Tool Used for Text Coding

4) The next step was to extract additional qualitative and quantitative information out of the original diary notes for every data entry within the spreadsheet. For this, appropriate extraction criteria were defined for each tag category. Then, every data entry was looked-up in the raw diary notes using the unique identifiers. Each of these text passages was then read thoroughly and information corresponding to the defined criteria was extracted into the spreadsheet. This sequence was executed for all tagged text passages. The most important information extracted was:
   - For each entry corresponding to a documented interaction or meeting with a project related person, the person's name, the duration of the interaction, and the topics discussed were extracted.
   - For each entry corresponding to a documented activity undertaken by the main software developer, the duration and the type of activity were extracted.

   Using this additional data extraction after the text coding, a maximum of information relevant to the research topics of this study was gained out of the diary notes. In order to estimate the accuracy of the extracted data entries about the developer's activities, a measure called the *data-gap* was estimated. The data-gap represents the percentage of work time per day for which no project-relevant information was found in the collected evidences (see also 4.1).

5) In order to avoid a possible methodological bias within the previous steps, this step consisted of establishing a second base of information out of the diary notes which is independent of the outcome of the previous steps. The previous steps focused on extracting quantitative data. In contrast, this step's outcome was mostly qualitative. It consisted of reading again through the whole diary notes and of extracting manually a list of key events in chronological order. Moreover, additional information, such as the persons involved in each event, was extracted. Furthermore, every concrete software requirement mentioned in the diary notes was also extracted and cross-referenced with previous occurrences. Thus, a written chronological summary of the project along with a chronology of the evolution of requirements was established independently of all the previous text coding and extraction steps. The newly gained data was joined to the previously established spreadsheet with the already extracted data. Through this, both independently gained data sets were cross-referenced by their timestamps for combined analysis purposes.
6) The following step consisted of crosschecking the spreadsheet established so far with all other collected forms of evidence. Notably, the following crosschecks were performed:
    - The datasets about interactions with persons related to the project were crosschecked with handwritten meeting minutes.
    - The data about the activities performed by the main developer were crosschecked using the detailed handwritten work-times table containing an entry for every single workday of the project.
    - The list of requirements and their evolution were crosschecked with the requirements documentation elaborated during the project. Moreover, the finally produced software package was inspected for requirements that were satisfied by the final product.

    This crosschecking step served as a boost in confidence in the accuracy of the collected data and it allowed the tight integration of the different sources of evidence into a single spreadsheet. Even the collected photographs mentioned in section 2.3.2 were used for crosschecking: The pictures of a whiteboard used for project planning purposes during the final project phases were crosschecked with the chronology of events reported in the diary notes.

7) The final step of the data processing approach was to organize all the information gathered within the spreadsheet for further analysis and to ensure that backtracking of every entry to the corresponding source is possible through the omnipresent timestamps. The information was clustered inside of the spreadsheet into several tables so that each table contained only information of a given type in quantitative form wherever applicable.

The fully indexed set of data contained within the spreadsheet database elaborated through the steps described above combines all relevant information collected from the various

sources. This corresponds to the concept of a "case study database" suggested by Yin [111 pp. 101-05].

The steps explained above describe the first part of the data analysis approach employed for this study. That part can be understood as a "pre-analysis" by which the "raw" collected evidence was transformed into a dataset with a higher degree of information density and accessibility: Only the information relevant to the research questions was extracted and made available for further analysis. The second phase of data analysis was then based on the spreadsheet resulting from the first phase.

The goal of the second phase of data analysis was to identify relevant facts that occurred during the project under study and influenced its outcome. Extracting facts from a data set containing already "condensed" information in a structured and accessible way was found to be very effective in combination with analysis approaches suggested by the literature. The following gives an overview of the methods taken into consideration for this study. Their outcome is described together with the analysis results in chapter 4.

The most important method employed for data analysis was based on the chronological sequence of events and the evolution of corresponding quantitative factors over time. This method is described by Yin as "time-series analysis" [111 pp. 122-27]. Yin differentiates between "time-series" in general, in which the evolution of quantitative values over time is investigated, and the special case of "chronologies", in which rather qualitative information is observed over time by focusing on causal sequences [111 pp. 125-26]. The latter method is described as the "compiling of chronological events" or "arraying of events into a chronology" [111 p. 125]. Both methods were used in the study for the analysis of the data.

The dataset established during the first analysis phase led to several types of chronological data series. Two types were especially relevant: the sequence of key events that occurred during the project and the emergence and evolution of requirements imposed on the developed system.

The first type, the chronology of key events, was found to be similar to the data analyzed in the widely cited case study of Walz, Elam and Curtis [109]. Similar to [109], the chronology established for the present study served to identify causal relations affecting the project's evolution and its outcome.

The second type of chronological data series is the evolution of quantitative data over time, which was analyzed using "time-series" techniques [111 pp. 122-25]. As described above, one of the outcomes of the first analysis phase was a chronological dataset accessible for quantitative analysis. The quantitative time series analysis was performed in the following way: All values of every category of quantitative chronological information were graphically plotted over time in order to make patterns visible and to draw conclusions about causality of influences. The graphical plots were produced by grouping several data fields belonging to the same category of a chronological series together into a two dimensional plot over time. By doing this, the following degrees of freedom inherent to the data were investigated.

- Accumulation of data values over time-periods: The frequency of meetings, interaction and activities that occurred during the project can be summed up in order to get, for example, the number of meetings per day, per week or per month. It is also possible to differentiate between calendar weeks or months and effective workweeks or work-months, by explicitly ignoring holidays. Furthermore, meetings and activities can be summed up either by their frequency of occurrence or by their averaged duration. Several different options were tried for the accumulation of data values before plotting.
- Aggregation of datasets: The different reported activities of the main developer can be aggregated into groups of activities. For example, the requirements elicitation, the requirements analysis, and the requirements documentation writing can be analyzed separately over time, or they can be aggregated together into a value representing the total activity related to requirements. Different ways of grouping datasets together were investigated.
- Types of plots: Different types of two-dimensional graphical plots were tested in order to find for every data category the type best suited to show the data's tendencies. Plot types like bar charts, line charts, bubble plots and area charts were tried.

These degrees of freedom, which were identified during the analysis of the data, are very common in case study research since the right arrangement of data can reveal far more insights than an inappropriate one. Often it is not possible to know in advance which arrangement is the best, and many variations have to be tried. Yin describes this as the approach to "play with the data" [111 p. 138].

The remainder of this section is dedicated to final remarks about the data analysis approach implemented for the present study.

It should be noted that the method of time series analysis might seem similar to the widespread case-study method of "pattern-matching", but this is not the case. According to Yin, "pattern-matching" is used to analyze the effects of dependent variables on collected quantitative data in order to reveal causal relations by comparing the empirically gathered data patterns to ones predicted by theory [111 pp. 116-20]. The time-series analysis method of the present study does not analyze the influence of dependent variables but solely investigates the evolution of variables over time.

Finally, it should be mentioned that during data analysis the identities of the persons who were part of the project were made anonym for reporting purposes. The real names of the participants naturally occur in the raw data material and within the pre-analysis spreadsheet. The real names were deliberately not used. Revealing the specific identities of the person would not bring any additional value to the study (see also [111 pp. 157-58]).

2.3.4 Threats to Validity

Considering and explicitly addressing possible threats to the validity is essential for producing high quality case studies [111 pp. 19,33-39]. Therefore, this section investigates threats to

validity discussed in the literature that were found to be relevant for the current case. Moreover, the counteracting measures taken within this study are briefly explained.

The first and most evident threat to validity is the possible subjectivity (the personal bias) of the collected data. For a high quality case study, the evident goal is to have as much valid data as possible. Many case studies try this by emphasizing the objectivity of their evidence through the collection of as much quantitative data as possible. According to [86]:

> "*Numbers may be overemphasized*. In trying to avoid subjective data, it is easy to overreact. Quantitative data is more objective and repeatable than subjective data, and may therefore be a sounder foundation for action. But until you know what questions to ask and what to measure, an emphasis on quantitative data may be misguided." [86 p. 24]

Nevertheless, this approach does not seem convincing because it compares two distinct aspects of data, *quantitative* versus *subjective*, which are, in fact, two completely different dimensions orthogonal to each other [99 p. 563]. It seems more natural to oppose *quantitative* data to *qualitative* data on one hand and to consider *objectivity* and *subjectivity* of data separately on the other hand. In order to clarify these two separate dimensions, it should be first noted that the notions of "qualitative" and "quantitative" do not only apply to the collected data. They are also used to differentiate between qualitative and quantitative empirical research [110 p. 8]. While case studies can be described as "research in the typical" and are not constrained by laboratory conditions (see 2.1.1), they can have both qualitative and quantitative research aspects [110 p. 10]. This is also in accordance with Yin, who describes the basis of case studies to be independent of the qualitative or quantitative nature of the data [111 p. 14]. In [112 p. 58], Yin affirms: "Case studies can be done by using either qualitative or quantitative evidence."

To give quantitative data in empirical studies any preference over qualitative data is, therefore, not justified. A clear statement about this issue was found in [50 p. 32]:

> "In general, as any other scientific domain, we should keep in mind that *empirical studies are a means, not an end*. Thus, we should pay more attention to their significance and contribution, and not just to the quality of the experiment design or, worse, to the amount of statistical curve fitting. Moreover, we should not automatically disqualify as "non" scientific those efforts that are not based on statistical evidence and controlled experiments. […] Some of the most important contributions in computer science were not based on empirical studies (as we define them today) and statistical evidence."

Finally, an excellent summary clarifying the controversy between qualitative and quantitative data in empirical studies was found in [99 p. 563]:

> "[…] the distinction between qualitative and quantitative data has to do with how the information is represented, not whether it is subjective or objective."

More concretely, Seaman [99] confirms that the coding method used within the first phase of data analysis of this study "[…] transforms qualitative data into quantitative data, but it does not affect its subjectivity or objectivity." [99 p. 563].

The threat of subjectivity for the present study can, therefore, only come from the data itself and not from the analysis approach taken. As described in section 2.3.2, considerable effort was put into the collection of several different sources of evidence, of which some were artifacts resulting from the investigated project and, therefore, are "naturally" objective. The subjectivity of the diary notes was minimized by focusing on a primary level on the reporting of meetings, activities, and events with as little bias as possible. To enhance objectivity, facts extracted from the diary notes were crosschecked as far as possible with information from the other sources.

Besides the most evident threat of data subjectivity, there are other threats possibly affecting the validity and quality of a case study. In order to discuss them systematically, the categorization of threats to validity elaborated by Yin [111] was taken as a reference. Yin [111 pp. 34-37] defines the following types of threats, which can also be partly found in [82 p. 351]:

- Threats to *construct validity*
- Threats to *internal validity*
- Threats to *external validity*
- Threats to *reliability*

In order to address the relevance of each of these types of threats, the following discusses each type of threat according to its characteristics described by Yin [111 p. 34]:

- *Construct validity* concerns mainly the correctness of operational measures employed for data collection. The approach taken for data collection, as described in section 2.3.2, shows that explicit attention was given to collect data from as many different sources as possible and to perform crosschecks between then.
- *Internal validity* is only relevant for case studies having an explanatory character and affects mainly the analysis phase of the study. The present study contains besides descriptive also explanatory elements as it tries to explain which causes led to the specific issues encountered during the project. As described in section 2.3.3, special precautions were taken by applying an elaborate data analysis approach.
- *External validity* deals with the capacity of the study's results to be generalized from the single observed case for any other comparable cases. Yin suggests *theory building* and the analysis of multiple cases as counteracting tactics. Performing a multiple-case study was, unfortunately, not feasible within the imposed constraints. Instead, theory building based on findings from the analyzed data was considered for this study.
- Finally, *reliability* addresses the necessity for the operations of a case study to be repeatable and concerns, therefore, mainly data collection, data analysis, and reporting of the employed methods. As shown by the previous sections, this case study report extensively explains the approaches and methods employed during the study. Moreover, the operational context of the study is thoroughly explained. Together with

the case study database of collected evidence, both tactics suggested by Yin to boost reliability are realized.

Turning from threats to validity towards threat-related quality measures, one can again consider Yin's suggestions. He defines explicitly quality measures for data collection by listing weaknesses inherent to common sources of evidence [111 p. 86] and sets recommendations for conducting high quality data analysis [111 pp. 137-39].

Concerning quality issues of data collection, the most common issue of data sources is their possible "bias" and the "selectivity" of the data collector [111 pp. 85-97]. The issue of a possible bias in the collected data for the present study has already been addressed above. The issue of selectivity has two aspects. First, the selectivity of the object under study is a potential issue and second, the selectivity during the collection of data. The first aspect of selectivity is not relevant for the present study since there was no "population" of projects to choose from. This issue is relevant for studies where the researcher can pick one or several objects for his study out of a pool of potential objects. In the present case, the project was given by the imposed constraints. The second aspect of selectivity, namely the selective collection of evidence, is relevant for the present study. It has been addressed by putting as much effort as possible into the collection of all relevant evidence without filtering it (see section 2.3.2).

Finally, the possible quality issues with data analysis were tackled by considering Yin's suggestions for high quality data analysis [111 pp. 137-39]:

- *Consider all collected evidence*. This was done by performing data analysis in two phases. The first pre-analysis phase served as transition between the variety of raw data sources and a complete database containing information from all sources using coding and crosschecking.
- *Address rival explanations*. Possible alternative interpretations were taken into account during the interpretation of the data.
- *Address the most significant aspects only and do not get lost in irrelevant details*. The approach of pre-analysis was taken to filter the information relevant to the research question out of the collected data.
- *Use specific and repeatable methods for analysis*. Different specific methods were employed, based on suggestions from the literature (see section 2.3.3).

All these precautions and considerations described above show that effort was put into learning about possible quality issues of case studies and applying counteracting or preventive measures whenever possible. Most "good practices" followed were essentially based on Yin's work [111], which is a well-known reference for case study researchers.

3 PROJECT PRESENTATION

This chapter describes the project on which the author worked for eleven months and which defines the case under study. Hence, this chapter serves two purposes. On one hand, it provides a detailed description of the context and the project's characteristics for the case study. On the other hand, its purpose is also to describe the work done by the author in his role as main developer on this project.

The chapter is organized into two parts. At first, the context of the project is described by focusing on its specific aspects. Secondly, the outcome of the project is presented by describing the solutions that were developed during the project.

3.1 Context of the Project

The project under study was influenced by its very specific context. By being conducted in an international research organization, it shall first be described in which setting the project was executed. Secondly, the very specific aspects of the organizations' technical infrastructure that influenced the project shall be explained. Furthermore, as the project evolved around the redevelopment of the software package for an already existing system, this system is described as well. Finally, the previously implemented software solution is presented as it served as a basis for the project and is thus a relevant part of its context.

3.1.1 Organizational Context

An overview about the organizational context of the project is given before describing the concrete organizational unit in which the project took place.

3.1.1.1 Overview

CERN is subdivided into several departments of which the "accelerator and beams department (AB)" (restructured into the "Beams Department (BE)" [29]) is dedicated to the development and maintenance of the technical controls infrastructure of the particle

accelerator complex [26]. Due to this mandate, the AB department holds all of the technical key competences necessary to conceive, develop, deploy, and maintain the particle accelerating machines and their surrounding infrastructure.

Projects related to the control systems of the particle accelerators have to deal with all aspects and conventions of the present infrastructure to allow seamless integration into the overall accelerator complex. These aspects are the key competences of the "Controls (CO)" group within the AB department. Within the CO group, different sections are responsible for the different aspects of the accelerators controls infrastructure. At this organizational level, the project under study took place.

Unfortunately, the project under study was not in line with the main business of the section in which it took place. The reason why this project was nevertheless conducted within this section was that the section leader had been one of the former developers of the system's hardware. He took the responsibility for the system's software and electronics hardware with him into his new position as section leader. Hence, it was decided that the responsibility for the system should be handed over to another group within the AB department whose activity and scope matched very well the system's domain. This group would become officially responsible for the whole system. Before handing over the responsibility, it has been decided that the system should be fully functional with initially foreseen capabilities that had not been implemented during the initial development of the system.

The machine as well as the hardware of the system of this project had been developed and integrated into the accelerator infrastructure already four years before the project under study took place. A first software package for the system had already been developed together with the initial system development. This initial software satisfied the most important user requirements at that time but left some needs unaddressed. These needs were based on the fact that the machine and its hardware had more potential capabilities than the software allowed its users to exploit. Nevertheless, the overall system was in use since that first development effort despite of its provisional state. Because it satisfied the high priority needs, there was initially no incentive to make changes to the initial software solution. However, for several reasons, the present project was launched with the goal to fully redevelop the higher levels of the system's software and to improve parts of the lower levels of the software. Besides the mentioned responsibility issue, also technical reasons explained below led to the redevelopment project.

3.1.1.2 The Project's Organizational Unit

From the organizational unit in which the project was conducted, five persons took part in it: the section leader (SL), the former developer (FD) of the system's initial software package, the main software developer (MD) of this project, and two experts that were only partially involved. One of these two experts was the low-level frontend software expert (FE) whose expertise was the development of system drivers for the employed embedded systems with the LynxOs operating system. The other expert was the high-level middleware expert (HE) who had in-depth knowledge and experience with client side of the CERN controls

middleware. Additionally, one person from the group that would take over the system's responsibilities took part in the project: the machine expert (ME). His role can best be described as being the expert for the machine controlled by the system and at the same time being the only person of the project who had knowledge about the end user requirements.

The six different persons involved in the project are described in closer detail in order to given a complete description of the organizational context of the project. The roles and relations of the project's participants are relevant for the case study as they had influence on its progression and outcome.

The section leader (SL) was responsible for the project within the organization and was the leader of the organizational unit in which it was conducted. Moreover, the SL had been part of the team that developed the specialized electronics of the system hardware during its initial development. Therefore, he was the main source of knowledge about the electronics part of the system's hardware. By being part of the original development team, he also had valuable knowledge about some parts of the domain and was a source of requirements concerning several aspects of the project. He was responsible for the availability and quality of the electronics and the software at the time of handing it over.

The machine expert (ME) was part of the group that had to take over responsibility for the system after the end of the project. The ME had already participated in the initial development of the system's custom hardware. By being an expert for the underlying physics of the machine, which is controlled by the system, he was an important source of domain knowledge. He partly had the role of a user representative, because he was the only participant who knew how the system would be used by the organization's machine operators. The real users would be a group of operators supervising the overall accelerator complex of CERN from a central control room. Unfortunately, these operators were not accessible for this project. Notably, the ME himself, as an expert for the machine physics, would be the user of some "expert" functionality of the system. This functionality was mainly related to machine maintenance and tuning purposes. It should be mentioned that the ME had apparently had issues with the SL at occasions predating this project. These difficulties in their relation emerged during the project but the originating causes remained hidden for the empirical study. Nevertheless, being aware of the presence of such tensions helped to understand some communication issues experienced during the project.

The former developer (FD) was one of the most relevant participants for this project. He had been the main developer of the first software for this system during its initial development. As a former developer, he had valuable domain knowledge about its internals and was a potential source for regaining the undocumented requirements that were relevant during that previous development effort. Moreover, his later activities made him an expert for parts of the organization-wide middleware system, which also played a major role for this system. The initially developed software for the system has been the FD's first project after having joined the organization. Unfortunately, despite being an essential member of the project, the FD had other tasks on his duty that had higher priority than this redevelopment project.

The main developer (MD) is also the author of this thesis. He joined the organization exclusively to work on this project. He had no previous experience with developing software in this organization. His task was the full development work of the project: requirements engineering, design, implementation and testing. He was the only person working exclusively and full time on the project during its whole timeframe. He developed the majority of the new software solution that was the project's outcome. Especially the three new Java-based client applications (see Section 3.2) consisting of over 17,000 lines of code were developed exclusively by the MD.

The lower-level software expert (LE) was part of the project only during the last third of the project's time-frame. Initially, he was not affiliated with the project in any way, but he was a member of the section if which it was conducted. He had expert knowledge about the low level of machine control software in general that was necessary for this project. He was not the only source of this kind of domain knowledge but he had the most practical experience with it. He was asked by the SL to join this project so that it could benefit from his domain knowledge.

The higher-level software expert (HE) became part of the project only during the last third of the project duration. Like the LE, he was a member of the section in which the project took place and he was asked by the SL to bring his expertise into the project. He had essential domain knowledge about designing and implementing client applications for control systems within the organization. This knowledge originated from several years of experience with the organization's conventions for such software systems. Moreover, he had in-depth experience with the client side of the organization's custom middleware. This domain knowledge was not available in explicit form from any other source. It was important to build a system that satisfied the organizational conventions so that it could be accepted by the operators. Therefore, it was vital for the success of the project to get a share of the HE's expertise.

In addition to the organizational context, also several technical aspects strongly influenced the project. Notably, a change in CERNs middleware triggered the reimplementation of parts of the system's software.

3.1.2 Technical Context

The following description of the technical context of the project focuses on its hardware part and its software part, as they both define the infrastructure of the organization.

3.1.2.1 Hardware Context

CERN was founded as a research center for particle physics and from its very beginning its main task was to conceive, construct, and maintain the infrastructure necessary to conduct scientific experiments within its research domain [31, 9]. The major part of this infrastructure consists of particle accelerators for the production of high-energy particle beams for physical experiments. Notably, two particle beams accelerated into opposite direction in circular

accelerators are brought into collision inside experimental facilities that capture the resulting products of the collisions.

During its more than 50 years of history, several different particle accelerators have been built at CERN, forming the central part of the physical research infrastructure. An overview of the most important parts of this infrastructure is depicted in Figure 2 (based on [51]). The figure shows schematically the different circular and linear accelerator machines together with their respective abbreviated names. The depicted outline of the infrastructure indicates the diverse interconnections between the different accelerators that allow synergy effects.

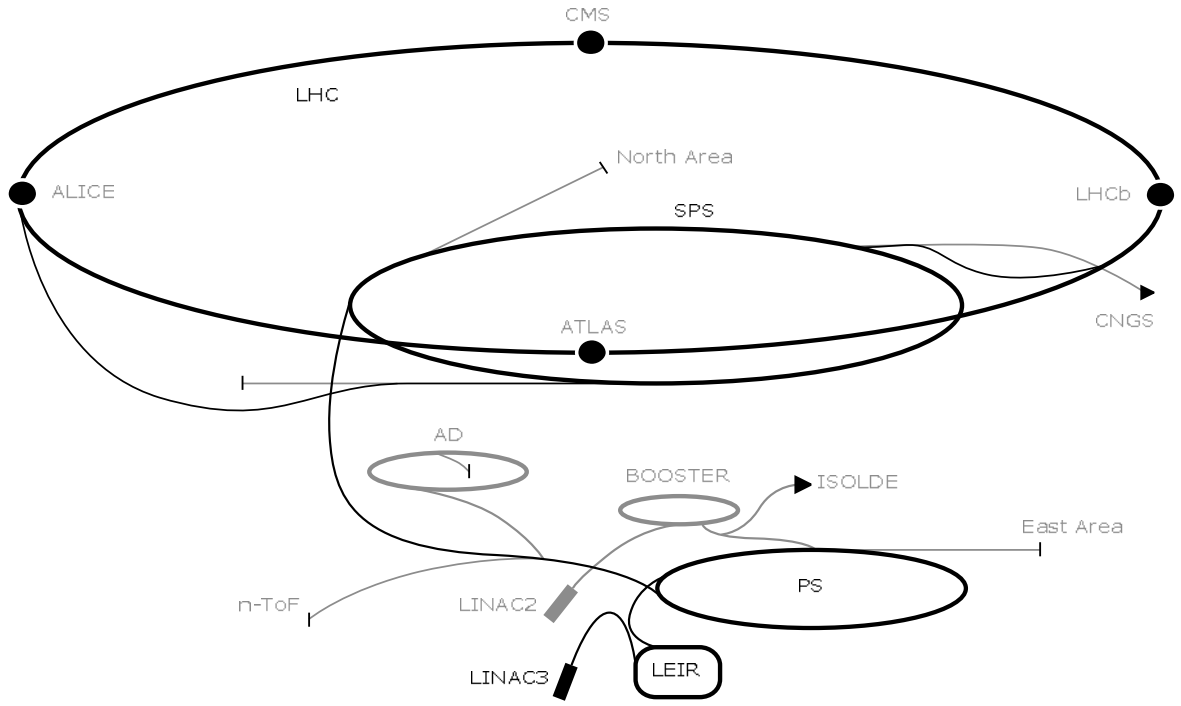

Figure 2. The CERN Accelerator Complex (based on [51])

For the majority of CERN's experiments at the present date, particles are generated and pre-accelerated into bunches of particle beams either in the LINAC2 or the LINAC3 linear accelerators (see Figure 2). Starting from there, the particle beam bunches are handed over from one circular accelerator to the next. In each of these accelerators, the particle beam is accelerated and energy is increased until a desired level of energy is reached. The beam is then extracted at experimental exit points or is brought into collision with another beam inside particle detectors.

The biggest experimental particle detector facilities of CERN are at the present date located around the Large Hadron Collider (LHC), the biggest particle accelerator to this date [98, 94]. The LHC can operate with two types of particle beams: Proton beams originating from LINAC2 and heavy lead ion beams originating from LINAC3.

The project on which this case study is based on was related to one of the subsystems of the LINAC3 lead ion accelerator. The development effort conducted in this project was necessary to enable future experiments with heavy lead ion beams with the recently inaugurated LHC accelerator. First experiments of this kind were run on the LHC in 2010 [24]. During such experiments, accelerated ion beam bunches are brought into collisions in several experiments located around the LHC in order to study physical conditions close to those estimated during the big-bang at the creation of the universe [98, 94].

As explained below, the main goal of the investigated project was to enable the low-level radio frequency cavity control system of the LINAC3 to work in a so-called “Pulse-to-Pulse Modulation (PPM)” mode (see [19]) and to offer a new software control interface for this subsystem to the operators in the CERN control room (CCC). Moreover, this new software had to be fully integrated in the accelerator controls infrastructure.

A large number of hardware control systems in CERN's current infrastructure are interconnected through the so-called “Technical Network (TN)”, which is based on gigabit Ethernet [49]. Each of these control systems, called “frontends”, hosts a real-time operating system providing basic Ethernet connectivity. The real-time system LynxOS is currently in wide use at CERN for this purpose [49]. Also the frontend system of this project was equipped with LynxOS. On top of the operating system, a number of software layers are employed to provide higher communication services.

In addition to the gigabit Ethernet network used for machine data exchange, CERN has a dedicated, custom-made timing distribution network, which offers essential services to all of CERN's particle accelerators. Because of its importance and strong influence on the requirements of this project, the CERN timing system is briefly explained.

The timing system is currently based on a dedicated physical network infrastructure consisting of special cabling interconnecting proprietary timing master and slave systems in a multi-drop RS-485 network [100]. The purpose of this timing distribution network is to get synchronous information about the current time at every frontend system in the widely dispersed particle accelerator facilities. Since accelerators like the LHC operate with particle beam bunches circulating at nearly the speed of light [98], a purely reactive, event-based system would not be feasible. Instead, the frontend systems are configured ahead of time through the data network and are then working synchronously based on the calibrated timing impulses from the timing network.

The solution offered by the timing system is to regenerate locally at every frontend system timing impulses that are synchronous within a nanosecond all over the dispersed facilities [100]. The frontend systems use these impulses to process previously configured actions at predefined points in time. The corresponding configurations are preloaded into the systems ahead of time usually by using the technical Ethernet network. This timing system is of such importance in CERN's infrastructure because it imposes a synchronous way of working for far dispersed but physically related machines.

The basic time unit in CERN's system is the so-called “cycle”, which has a duration of 1.2 seconds [70]. The configurations processed by every frontend can be changed from one cycle to the next or after a series of such basic cycles (called “super-cycle” [67]). The series of cycles played by the different accelerators in order to produce a particle beam with certain characteristics for a given experiment or “user” within the accelerator complex is called a “compound cycle” [70]. Out of a defined compound cycle, the cycles on different accelerators, that do not interfere with each other can be executed simultaneously. This principle allows time division multiplexed use of the particle accelerators, which minimizes idle times and allows an efficient use of the facilities. Some systems needing finer grained timing impulses can produce these locally, based on the global synchronized timing pulses. These essential aspects of the timing system were essential design drivers for the project of this study.

An example shall illustrate the purpose of this cycle-based approach based on the CERN accelerator complex as depicted in Figure 2:

> “As an example, the Compound Cycle to make a beam for SPS fixed target physics begins with protons in the linac, they are then accelerated in the PSB, a further acceleration in the CPS, then in the SPS, and finally, the beam is ejected to the fixed target user.” [70 p. 15]

This hypothetical example of experimental runs is in reality more complicated. Complex handovers of particle beams through beam pipe interconnections can be performed by this flexible system, as all parts of the complex work in a synchronous manner based on the basic 1.2-second cycles. It allows almost non-stop runs of the accelerator facilities. This “way of working” imposes technical requirements on all frontend systems and represents, therefore, an essential part of the technical context of hardware and software development projects within this domain, such as the project under study.

3.1.2.2 Software Context

Projects at CERN dealing with any part of the accelerator facilities have to cope with a complex special-purpose software infrastructure. This “software context” has an important impact on a project's requirements and its design. Under such circumstances, the specific domain knowledge plays an essential role. This specific software context can be explained by looking at its distinct parts.

Software Building and Deployment

At CERN, there are currently two major software building, versioning, and deployment systems in use for accelerator controls related projects. The first system concerns graphical user interface applications on the client side, which use CERN’s middleware and specialized utility libraries. This development is based on the Java programming language. The other system concerns the middleware libraries and the application programs running on the server side, i.e. on the frontend systems themselves. Those are written in the C/C++ programming language.

The first of these two systems consists of a concurrent versioning server (CVS) server, a "release tool" with a build server, and a special Ant Script called "common-build" [102]. The parts of this system are tightly related to each other. Each project developed with this system needs to have a unique project name, that has to follow a hierarchical naming convention. The project under study was identified as "accsoft-rf-lrfsc". The common-build script helps the developer in resolving dependencies to CERN libraries and in locally building an application. In order to compile, deploy and test accelerator controls software for a release, the following automated procedure had to be used:

1. The software developer checks in the latest version of his source code into the CVS server and tags it with a version tag (according to a convention).
2. The software developer launches locally on his machine the common-build Ant script that launches the build process remotely on the build server for the given version tag of the specified project. The "release tool" on the build server then checks out automatically the code from the CVS server.
3. In a next step, the build server checks all dependencies of the software project to be built. Those dependencies mostly relate to the specific CERN middleware libraries for projects related to accelerator controls. If successful, the server fetches the specified version of those libraries for compilation.
4. The build sever then compiles the project according to the specified options and reports the progress remotely to the developer's client computer. Moreover, the build server can also compile documentation and run automated tests.
5. If the build is successful, the server uploads the produced binaries to a dedicated deployment server. This deployment server is accessible by all authorized clients inside the CERN network. The binaries built are packaged in a JAR file.
6. The build server and the local script terminate. The client control application can be launched through a JNLP file directly from the deployment server. This JNLP file points automatically to the latest version of the application. Libraries necessary for the launch of the application on an authorized client computer will automatically be fetched from the same build server. Client applications are, therefore, not deployed locally on the clients, but are instead directly run from the deployment server through the JNLP file. This avoids the use of any outdated version of a control application for the accelerator machines.

The second software development system concerns the part of the software running on the frontend systems themselves. It consists of a CVS server, a network file system (NFS) and build-scripts that are accessed either through remote terminal connections to console servers, or locally through the NFS. Through this, a developer can work directly on his C++ code in a directory shared by NFS and use the provided build scripts for the following tasks:

- Check out the project from CVS
- Compile the current code and build an executable for a specific frontend system
- Check in the source code for a project onto the CVS
- Deploy an application to a frontend system [56]

- When a developer has built the executable for his project, he can log on remotely to the target frontend system and launch the executable there directly using the NFS.

It is important to note that software development using this build system is tightly related to CERN's "Front-End Software Architecture" (FESA) [52], as explained below.

Software development in such a context is strongly based on proprietary conventions and tools. Consequently, an important amount of specific knowledge is necessary to successfully design, implement, and deploy software to a CERN accelerator frontend system. Hence, this knowledge is an essential aspect of the software development context of the project under study.

Software on CERN Frontend Computers

When developing software in the frame of a CERN frontend system within the accelerator complex, there are several levels of existing software infrastructure that have to be used and taken into account. Parts of this software have been developed at CERN for its specific purposes and are as such highly specialized. Its most important parts are the CERN Frontend System Architecture (FESA) and the "Controls Middle-Ware" (CMW). An overview of these software packages and its surroundings is given in the following.

Figure 3 (based on [23]) depicts the software at the different levels of a typical frontend system for accelerator controls and its interfacing surroundings. The predominant type of frontend systems currently in use at CERN for this purpose are VME based industrial computers running a commercial real-time operating system called LynxOS [49]. The software running on this operating system is represented by the layered cube in Figure 3. It is composed of two different levels of software as well as of the server-side middleware, each represented by different shades of gray: The low-level software (in white) concerns the interfacing of the VME cards. The high-level software (light gray) can be described as the "business logic" of the frontend system. Finally, the top part (dark gray) is the controls middleware, which communicates towards software beyond the frontend system boundaries. The three parts have the following characteristics:

- The low-level frontend software is the interface to the physical machine controlled by the system via specialized VME boards. The drivers have to be specifically developed for the employed VME I/O board. For the development of such drivers, knowledge is necessary about the VME system, the frontend computer system, the LynxOS operating system, and the VME board hardware. The driver is the first abstraction of the raw data interface to the dedicated hardware. As such, it hides the memory map necessary to access the boards through the VME bus. Towards the higher software levels, the low-level software offers an Application Programming Interface (API) to access the driver. In this API, Unix IOCTL calls are used to access the desired driver functionality. The development of this low-level software can be facilitated by CERN's DriverGen utility [23], but for the project under study, the development was fully manual.

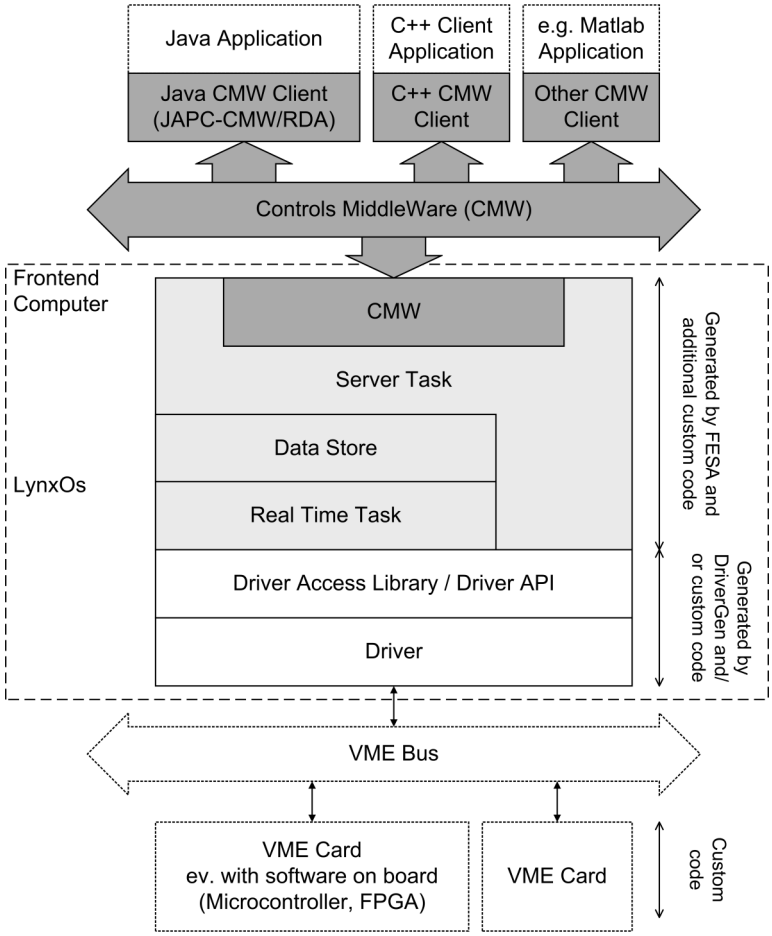

Figure 3. CERN Frontend System Software (based on [23])

- The high-level frontend software can be seen as the business logic of a frontend system. It processes so-called Real-Time (RT) tasks and server tasks by using a common data store. This software level relies on automated C++ code generation provided by the FESA framework. The FESA framework provides the frontend software developer with tools for designing, developing, deploying and testing the high-level frontend software [3]. Every frontend system controlling a given type of machine has a corresponding object-oriented model of the controlled machine parameters in FESA. Such a model is created using a Java-based FESA development tool called "FESA-Shell" and is stored in a database [56]. During software development, this model is retrieved by the software build scripts and is used to generate all necessary code for locally storing the machines control parameters (the data storage) as well as code for "server tasks" for the manipulation of the parameters remotely through the middleware. Additionally, code is generated for "real-time tasks" (RT-tasks) which are necessary to make the system react on events generated from CERN's timing system (for this, every frontend system is equipped with a CERN timing network receiver card) or other event sources. Based on the generated code, the frontend system developer adds code to the RT-tasks that reads and writes machine parameters to or from the local data storage and communicates them over the VME

bus to the specific hardware board at predefined instants of time. Moreover, the developer adds code to the server tasks, which processes machine parameters, communicated using the middleware, and reads and stores them in the local data storage. If necessary, server tasks can directly call the driver API to interface the hardware. Thus, the high-level frontend software is the system's core business logic as it makes the system react to timing events and processes control parameters passed between the controls middleware and the actual machine hardware through the driver interface.

- The controls middleware plays a central role for CERN's control systems as it makes all the different parameters from the frontend systems of various machine parts of a particle accelerator accessible in a unified manner. This allows the centralized control of the particle accelerators and its surrounding infrastructure. Because of its importance for software development efforts on the frontend and the client side, the controls middleware is described in the following section. On the frontend side, CMW uses the LynxOS' Ethernet capabilities to communicate over CERN's technical network. The CMW program code is automatically linked and interfaced through the FESA framework during the development of the high-level software.

In addition, Figure 3 shows the frontend software's two interfaces: the low-level interface towards the physical machine and the high-level interface towards the clients through the controls middleware. The controls middleware is described in the following dedicated section. The low-level interface is briefly described as follows:

In order to fulfill their specific control functions, the frontend computers are equipped with a wide range of VME-based input-output electronic cards adapted to each specific application. This is represented on the bottom of Figure 3. Many of those cards are custom built by CERN. In the case of this project, the special input/output (I/O) card, called LLCC (Low Level Cavity Control), is a digital electronic control card for radio frequency resonance cavities. The special I/O cards of such frontend systems control the physics of a machine to which the system is connected. Each machine is moreover integrated into the overall accelerator infrastructure of CERN. In the case of this project, a radio frequency resonance cavity, which is part of the LINAC3 accelerator, is controlled by the LLCC card.

Behind this low-level interface, outside of the frontend computer, there can also be additional specific software within the machine electronics. This can be specialized subsystems encompassing programmable microcontrollers or field programmable gate arrays (FPGA) as indicated in Figure 3. Such specialized subsystems have to be developed in order to satisfy the interface requirements of the low-level frontend software. This was the case of the system under study, whose design spanned all the levels of software described above down to a FPGA in the dedicated VME board. Such specialized software is not standardized within CERNs development process but is nevertheless an important aspect of the software context of a project.

The CERN Middleware

The middleware implementations at CERN change over time, as experience with current systems shows flaws and new requirements emerge. During the development of the LHC, CERN's control systems underwent considerable modifications to accommodate for the specificities of this new accelerator in its infrastructure. The present control system is called the "LHC era" [67] of control systems as it was developed with that new accelerator in focus and it was intended to unify the diverse control subsystems in use in the different parts of CERN's infrastructure.

For the "LHC era", the CERN "Controls Middle-Ware" (CMW) has been developed [62]. It is a specialized software infrastructure developed specifically for the purpose of machine control systems of particle accelerators. It can be seen as a software data bus interconnecting the accelerator control frontend systems for unified access [23]. The CMW is by itself not a specific software package. Instead, it is the specification of a middleware system for accelerator controls, which comprises different implementations. One implementation is the CORBA-based Controls Middle-Ware for Remote Device Access (CMW/RDA) which is based on an object-oriented device model [63]. Its most recent implementation in JAVA, called JAPC-CMW/RDA, is the currently preferred controls middleware at CERN and it was, therefore, employed for the project under study.

As the subsystem of LINAC3, for which this project took place, had been developed before the JAPC-CMW/RDA implementation of the LHC era was available, the initial implementation of the system's client software was based on a previous JAVA implementation of CMW/RDA. The major change was that the former client-side middleware did not take fully into account the timing system context. This change in the client middleware was, therefore, also a major trigger for the reimplementation effort conducted in this project.

The overall middleware concept of CERN is outlined in Figure 4 (based on [62]). It partly represents the different layers of the current controls middleware with some of the possible variations. The cloud in the center symbolizes that any vertical slice on the lower server side can potentially interact through a CMW implementation with any of the vertical slices on the client side. The gray shade highlights the elements that are part of the overall middleware infrastructure of CERN. The server side on the bottom of Figure 4 corresponds to the frontend systems (can be FESA-based or specialized equipment servers) or gateways to other control subsystems. On this side, the RDA server API must be implemented. The client side on top of Figure 4 corresponds to either graphical user interface (GUI) applications for the operator of the particle accelerator infrastructure, the Java control programs (left side of client block), or of specialized application for expert users of specific machines (right side of client block). Additionally, such expert applications can also be written as Java control programs.

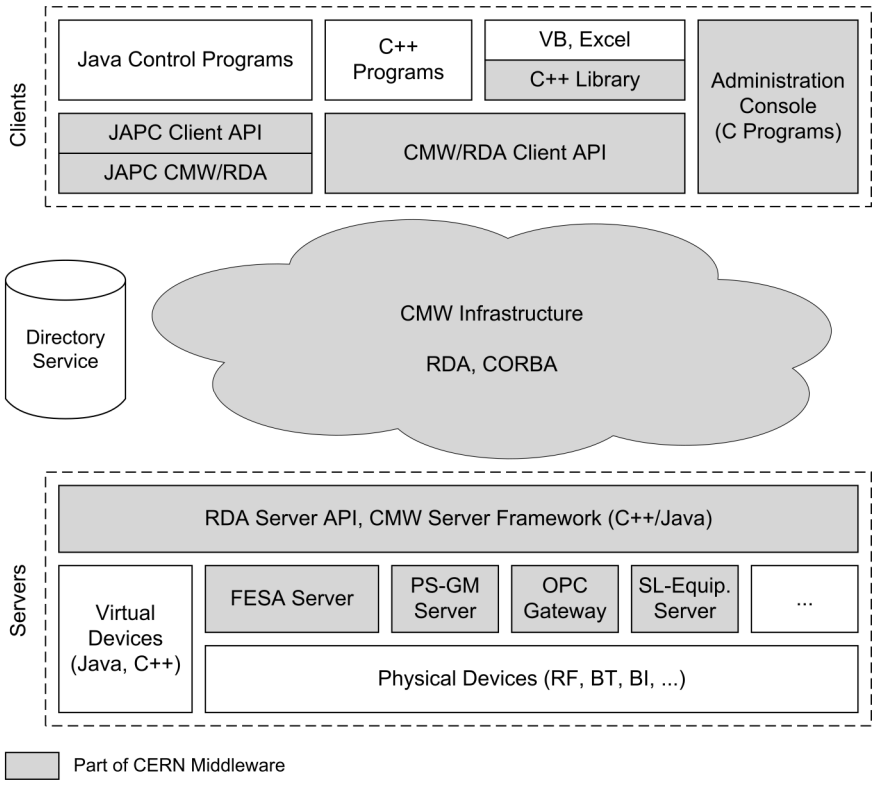

Figure 4. CERN Middleware for Accelerator Controls (based on [62])

To give an overview, the different elements of the project under study can be pointed out in Figure 4: On the server side, a radio frequency (RF) device was used, connected to a frontend system running a FESA-based CMW server. On the client side, three different Java control programs were developed using the JAPC CMW/RDA implementation. From these three applications, one was a specialized application dedicated to expert users but, nevertheless, it was also developed using the same JAPC CMW/RDA implementation to benefit from synergy effects while developing these applications interfacing the same CMW server.

The basis of CMW is the so-called device- and property-model of the different RDA implementations, which are based on CORBA [62, 63]. In that model, physical machines are referenced as “devices” with a device name as unique identifier. Each device in this model is an instance of a “device class” that specifies the “properties” accessible remotely through the middleware. These properties are accessible in an object-oriented manner through “get” and “set” procedures. As any kind of device controlled by a frontend system is susceptible to be used several times within one or more accelerators, it makes sense to define all the common properties as a “device class”. Every instance of such a “device class” can then have its specific parameters adjusted.

As the server-side software of a frontend system is generated by the FESA framework providing the server-side CMW implementation, the device and property model is built into it automatically. The “FESA Shell” tool provides a graphical user interface to the frontend software developer for specifying a device class with all its properties. For each property, data

fields are defined by unique identifier names along with their data types and default values. Moreover, FESA allows the definition of server side methods for remote procedure calls on a device class. These are called “server actions”.

For each property, default or custom getter and setter “server actions” can be defined for manipulating the data of device class properties. Custom getter and setter actions can be used to process data transformations between the middleware and the local data storage within the frontend. They are only generated as code stubs by FESA and have to be implemented manually by the frontend software developer. For example, such custom actions can be used to transform raw data values stored in the frontend data into physical units for remote data access or data plausibility checks can be performed. In the project under study, custom server actions were used to check parameter settings coming from the middleware for plausibility in order to avoid unintended machine behavior.

Besides the server actions, there are also so-called “real-time actions”, which are triggered by timing events coming from the timing system receivers in the frontend systems or other interrupt sources of the frontend computer system. Those actions are always generated as code stubs, as they need to be manually developed for each individual device class. The real-time actions are intended to set the correct control parameters to the device hardware via its driver at the correct time. Moreover, data from the hardware can be read-out to the local data storage at predefined timing events through such real-time actions.

On the client side, the typical application is a Java-based Graphical User Interface (GUI) interacting with one or several frontend systems by using a library offering a client-side CMW implementation. The predominant library is currently based on JAPC, CERN's Java API for Parameter Control. It is called JAPC-EXT-CMWRDA and is a CMW/RDA implementation for the client JAPC API. The JAPC API provides remote access to devices running CMW servers and adds support for the so-called “timing context” [67]. The client software developer develops the software as described above and uses the JAPC-EXT-CMWRDA Java library provided by the development environment when using CERN's common-build tool. This library allows remote procedure calls to a CMW device using its device name, the name of the parameter defined for the device's “device class”, the name of the data field in the parameter, and the timing context for which the parameter has to be accessed. The data field can thus be accessed using the getter and setter methods, which can be called from the client side in a synchronous or asynchronous way [5].

For a control application, it is useful to display changes of data in the frontend system to the user in a continuous manner. In order to avoid the polling of device parameters by a series of remote procedure calls to the get methods, a subscription mechanism is available in CERN's middleware. This mechanism can be activated from the client side by invoking a “MonitorOn” method on the CMW server for a given property and timing context. This will trigger update notifications from the server to the client on any subsequent changes of the property’s value. On the client side, a reply handler has to be implemented for processing property change notifications coming from the server. For the project under study, this subscription

mechanism turned out to be a major constraint for the design of the client-side software and even influenced some of the requirements for the system.

Another important part of CERN's middleware infrastructure is the directory service based on a naming and configuration database (see Figure 4) [5 p. 497, 41]. Since the frontend computer systems are interconnected using CERN's technical Ethernet network, a naming service for devices is employed to make the middleware communication between CMW servers and clients independent of the concrete TCP/IP configuration [62]. A running CMW server registers with the naming service and any CMW client accessing this server can query the server's hostname and thus its IP address.

The naming service is part of the directory service for accelerator controls, which can give client application diverse information about devices and device classes such as a list of available properties or timing system characteristics. Frontend systems as CMW servers using the FESA framework can also query the configuration database. FESA allows the specification of default values for properties and constant parameters for each instance of a device class (i.e., one specific device). When a given FESA based device boots up, it queries the directory service for its default property values. Thus, it is possible to deploy, for example, different instances of the radio frequency cavity control system at different parts of an accelerator and define specific location-dependent adjustments for certain properties (e.g., the exact cavity's resonance frequency).

The last important aspect of CERN's middleware infrastructure is its relation to the timing system. As mentioned earlier, CERN's dedicated timing system provides synchronous timing impulses distributed all across the infrastructure by using a special purpose network. Each frontend system is equipped with a Controls Timing Receiver (CTR) card, acting as a receiver in the timing network [69]. There is a centralized Master Timing Generator (MTG) system acting as sender on the timing distribution network [69, 8].

This timing system is closely coupled with the middleware, since the overall control system has to work in a synchronous way. On one hand, a frontend system receives information from its CTR card via FESA and triggers RT-actions. On the other hand, a client application connects through CMW to a frontend system and can subscribe for notification of value changes on certain timing events. Moreover, frontend systems which can adjust their configuration for every accelerator cycle, the so-called multiplexing [3 p. 311] or PPM mode [8, 19], need to have separate values of each multiplexed property for each specific cycle. Therefore, while getting and setting properties on a frontend system, a client application has to specify the timing context (in the simplest case the name of the cycle, i.e., the “user”) for which the remote procedure call has to be performed [62 p. 319]. The majority of the properties used in the project under study were such multiplexed properties.

3.1.3 System Description

The system for which the development effort took place in the frame of this project is called the “Low-level Radio Frequency Cavity Servo Control” (LRFSC). Using this system, the

amplitude and the phase of the electric field inside metallic resonance cavities can be controlled [101]. Bunches of particle beams passing through such cavities are accelerated or modulated by their interaction with the electric field. That makes such cavities and their control systems an essential component of particle accelerators. At CERN, several different systems of this type are in use. Depending of the type and the characteristics of the particle accelerator and depending on the time at which an accelerator has been built or upgraded, different cavities and different control systems are employed.

The LRFSC system was designed as a digital low-level radio frequency (RF) cavity control system in 2003 [93]. It was first employed for CERN's linear accelerator for heavy proton ions named LINAC3 for the low-level control of the energy ramping cavity added for the production of ion beams for the Large Hadron Collider (LHC) [101]. At the time of the project under study, several instances of the system were in use in LINAC3 as well as in two separate test facilities. Interestingly, all of these systems were maintained by the group that would take over the project, whereas the responsibility for the software remained with the section in which the project was conducted (see also section 3.1.1).

Figure 5 depicts an outline of the components that are part of the LRFSC system and its surroundings. The dotted area in the figure shows the frontend system of LRFSC, which controls the electric field of the cavity through a radio frequency signal amplifier, feedback signal pickups and the resonance control ("Res. Ctrl." in Figure 5). The system is interfaced through the technical network using the controls middleware (see section 3.1.1) and the frontend computer system's Ethernet adapter. Moreover, the system receives timing information from CERN's timing using the CTR timing card, which is added to the frontend computer system.

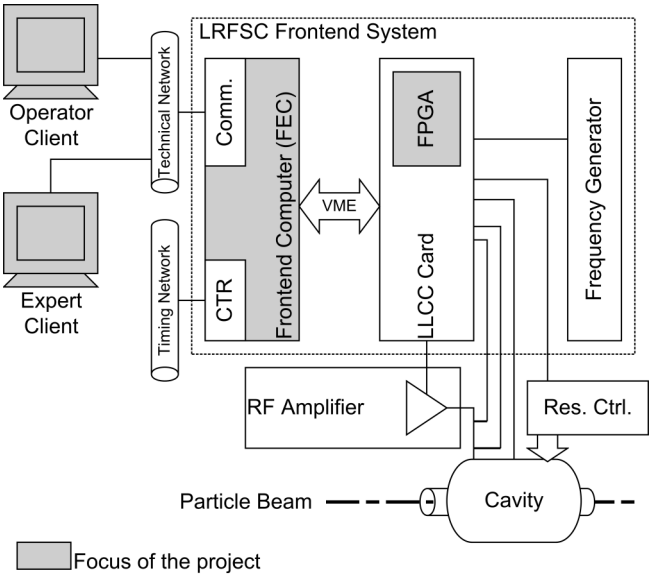

Figure 5. LRFSC Frontend System and its Environment

The heart of the LRFSC system is a custom made VME module developed especially for frontend systems controlling the radio frequency field in resonance cavities using digital signal processing. This module is originally referred to as the “Low Level Control Card” (LLCC) or simply as “LRFSC card”. Details about this card have been published in [93, 101, 20].

The LRFSC frontend system is composed of an embedded computer, the Front-End Computer (FEC), which runs an implementation of an LRFSC FESA class on top of LynxOS. This computer is connected to the LLCC through the system's VME bus as outlined in Figure 5. The LLCC interface over VME is defined as a memory map through which the embedded computer can read or write registers and memory arrays on the LLCC. The main component of the LLCC is a Field Programmable Gate Array (FPGA), which on one hand provides the memory map interface and on the other hand processes the servo control of the RF field. Moreover, each LRFSC system has to be equipped with a frequency generator for producing the modulation signal as well as synchronous signals with twice and half that modulation frequency necessary for the proper operation of the LLCC. In the case of the LRFSC test facility used for this project, these signals were provided by a generic frequency generator module integrated into the VME rack.

As explained in Section 3.1.1, the LRFSC system has been developed before the current project took place. The focus of the project under study had the purpose to fully redevelop the client software, as well as parts of the frontend software. In Figure 5, the parts of the system that were in the focus of this project are highlighted: the control applications for the clients (left side of Figure 5), the software on the FEC, and the programming of the FPGA on the LLCC.

The following shall give an overview of the LLCC and its interfaces as it had a major influence in the form of requirements and design constraints. For this purpose, a schematic outline of the functional blocks of the LLCC is shown in Figure 6 (see [93, 101] for further details).

In Figure 6, the different inputs and outputs of the LLCC are sketched around its dotted border. The central part is the FPGA, which interfaces the frontend computer through the VME bus shown on the left side and the cavity of the machine through the analog RF inputs and outputs shown on the bottom side. There are three input channels named Cavity (CAV), Forward (FWD), and Reflected (REF), and there is one output channel named Output (OUT). The latter is driving the RF field in the cavity through an amplifier as indicated in Figure 5. Moreover, there is a resonance control output providing a continuous analog voltage, which can be fed to a separate Programmable Logic Controller (PLC) for the purpose of driving an electrical motor that can mechanically deform the metallic cavity to fine-tune its resonance characteristics. Furthermore, the card provides analog outputs for diagnostic purposes (shown on the top of Figure 6). Digital signals occurring during the processing in the FPGA can be redirected to these four "diagnostic channels". Finally, the LLCC has dedicated inputs for clock and control signals (one on the bottom left and two times three on the top in Figure 6). One analog input is dedicated to the RF modulation signal (“Mod.Sig.” in Figure 6) which is necessary for the analog RF to digital signal conversion. Three digital inputs are dedicated for the clocking of the logic circuitry (FPGA, ADCs, DACs, RAMs). Another three digital inputs are

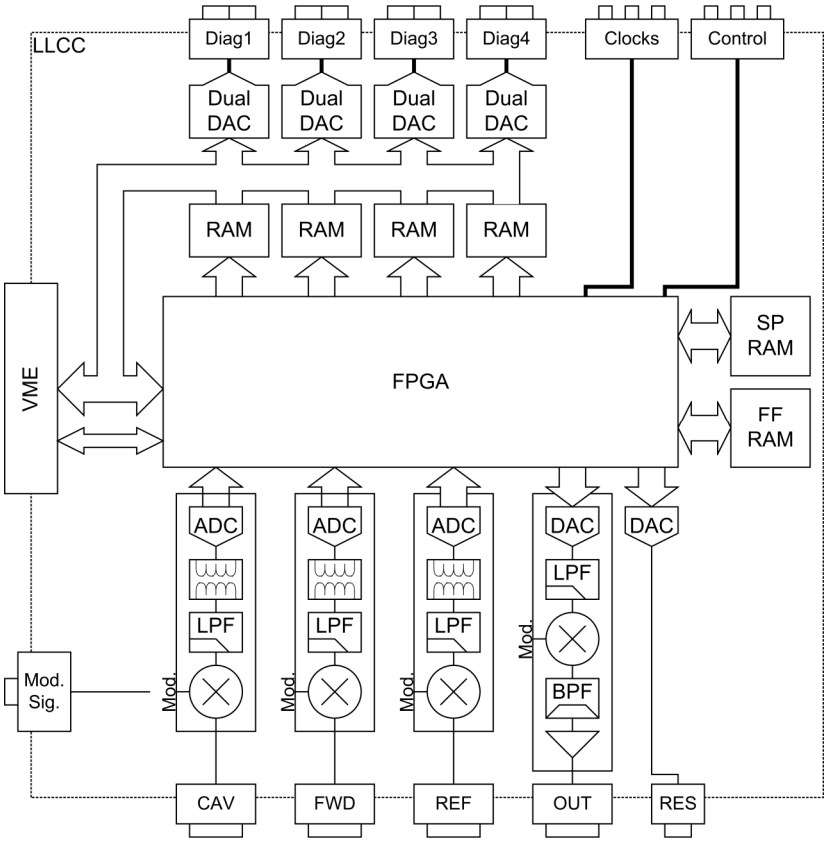

Figure 6. Functional Blocks of the LLCC

foreseen for acquiring control signals from the machine surroundings or a timing receiver like a precisely timed starting pulse.

Based on this description of the outline and the interfaces of the LLCC, a brief overview can be given of its operation principles.

In short, on the LLCC the "[...] FPGA implements a PI digital controller, digital modulator and demodulator, diagnostics and logging as well as the VME interface" [101]. To perform this signal processing, the incoming and outgoing signals need to be converted between digital and analog form. For this purpose, the LLCC uses a special conversion mechanism suited for RF signal processing called I/Q modulation and demodulation. For each input channel, the analog RF signal is down-converted to a tenth of the cavity's resonance frequency by using an external modulation signal ("Mod.Sig.") adjusted to the digital sampling frequency [93, 20]. In the case of the LINAC3 ramping cavity, the cavity's frequency is 202.56 MHz. This operation preserves the amplitude and the phase information of the original signal [101] and is, therefore, well suited for this low-level RF control. After the signal passed through Low Pass Filtering (LPF) and electrical decoupling, it is then sampled by an Analog-to-Digital Converter (ADC) at a sampling rate exactly four times the frequency of the down-converted signal. Using this sampling scheme, the sampled ADC values can be interpreted as a sequence of "[...] in-phase (I) and quadrature (Q) components [...]" [101], which correspond to the rectangular

coordinates of the phasor defined by amplitude and phase of the signal. This digital I/Q stream can then be used by the programmed logic of the FPGA to modulate the amplitude and phase of the signals. For this, the LLCC is equipped with two Random Access Memory (RAM) banks where a series of I and Q values can be stored, which adjust the modulation by being added to the I/Q data stream before (the Setpoints "SP RAM") and after (the Feed Forward "FF RAM") the digital Proportional-Integral (PI) controller [101]. After the complete digital signal processing, the output I/Q stream is converted back to an analog RF signal using the same principles as for input conversion. As depicted in Figure 6, the I/Q stream from the FPGA to the OUT channel is converted to analog I and Q values which are filtered, multiplexed with the modulation signal, filtered again and finally amplified for output. The overall procedure reconstructs an RF signal at the cavity's resonance frequency that has the desired amplitude and phase modulations imposed by the signal processing.

The next important principle of operation of the LLCC is the pulsed processing. Particle beams in CERN accelerators are usually formed by a sequence of short particle beam "bunches" per cycle. This is also the case for the LINAC3. Therefore, the LRFSC system and hence also the LLCC, need to work in a pulsed mode. More specifically, the modulation of the amplitude and phase characteristics of the RF field in the cavity (defined by the SP and FF RAM values), needs to be synchronized with the arrival of every beam bunch inside the cavity. To achieve this, an external signal, called the "RF-ON pulse", is used as an input to the LLCC. That signal, coming from the surrounding machine infrastructure or from a calibrated timing receiver, triggers the start of the signal processing by the FPGA.

In addition to this pulsed mode, the CERN accelerators work in time slots called cycles as described in section 3.1.2. Another digital signal is, therefore, fed to the LLCC indicating the start of a cycle. Since the SP RAM, the FF RAM, as well as the diagnostic RAMs are in use by the FPGA during the signal processing of pulses, their content cannot be accessed by the FEC through the VME bus at that time. Therefore, the end of an RF-ON pulse is used by the LLCC to send an interrupt signal to the frontend computer, indicating that RAM data can now be accessed for a short period before the next RF pulse starts. Combined with the knowledge about the number of pulses per cycle, it is possible to delay all time consuming RAM manipulation at the time after the last pulse in each cycle where a longer pause is made in the case of LINAC3. This fact strongly influenced requirements and design constraints for the frontend software and highlights the difficulties encountered during the software development.

Another important functionality of the LLCC is given by the diagnostic channels. During signal processing, the FPGA can redirect any I/Q data stream, like the incoming CAV signal for example, to one of these channels. Each diagnostic channel consists of a RAM module and a Digital Analog Converter (DAC) providing the same signal stored digitally in the RAM, as analog output for visualization on an oscilloscope, for example. The digitally stored diagnostic signal can also be retrieved by the FEC directly through the VME bus under control of the FPGA. As indicated above, it is only possible to access RAM data through the VME when no signal processing is done. When processing is paused, the FPGA disconnects its clock signal

from the diagnostic RAMs and connects the VME bus clock for direct access. Through a proper software package consisting of a server side in the FEC's FESA class, and a client application at a user's workstation, a remote digital storage oscilloscope functionality is possible.

This overview of the basic parts and principles of the LRFSC system defines the technical context in which the project under study operated. It emphasizes the diversity of domain-specific knowledge that was relevant for the proper execution of the project.

3.1.4 The Previous Software Solution

During the initial development of the LRFSC system, software for the frontend system has been developed as well as two client applications. The software solutions developed for both sides, the client's and the server's, is described below in order to show what prerequisites were available for the redevelopment effort of the project under study and to show the differences towards the newly developed software package (described in section 3.2). This initial set of software has been developed by the former developer (FD) in collaboration with the section leader (SL) for the lower layers of the software.

In Section 3.1.2.2, the general layers of software for control systems at CERN are introduced. Based on this general outline, the specific software layers of the initial LRFSC software implementation are represented in Figure 7. The representation also shows the interfaces of the layers, including interfaces towards hardware, which are differentiated by dotted lines.

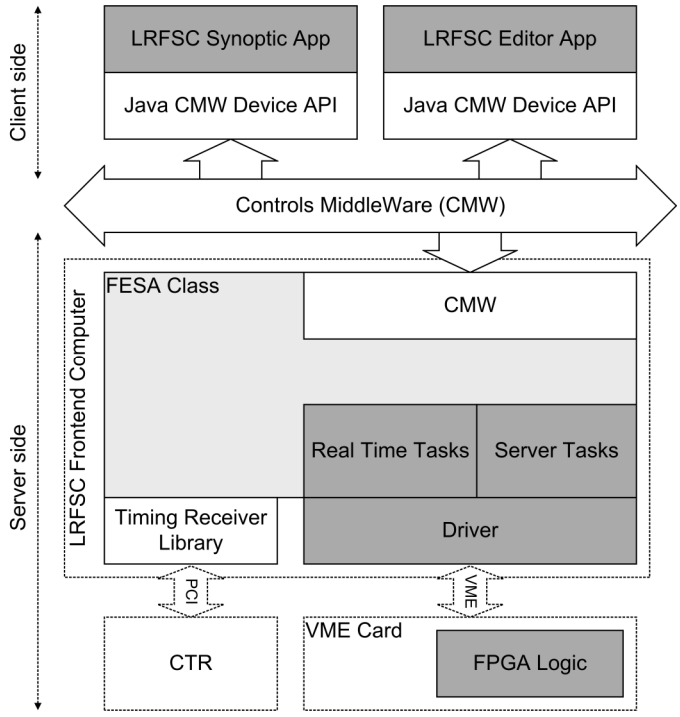

Figure 7. Software Layers of the Initial LRFSC Implementation

On the server side, a FESA class named “LRFSC” has been defined and server tasks as well as real-time tasks were implemented based on FESA framework version 2.9. Together with the FESA class, a LynxOS system driver for the LLCC has been developed (by the FD) as well as the VHDL code for the signal processing inside the LLCC's FPGA (by the SL). All those developed software parts are highlighted in gray in the lower half of Figure 7. The FESA class is light gray because it was defined and generated using the FESA shell tools in contrast to the other parts that were designed and coded manually. As indicated on the bottom of Figure 7, the real-time tasks and the server tasks directly call functions from the LLCC driver using "IOCTL" calls. The driver had the purpose of transforming these function calls into VME read and write operations from and to the LLCC's FPGA logic using a defined memory map. The FESA class' generic code triggers the server tasks for incoming CMW remote procedure calls. It also triggers the real-time tasks on given timing events received using the CTR timing receiver card f the frontend computer via its timing library (lower-left part of Figure 7).

On the client side, two separate Java client applications had been developed by the FD: a synoptic application mostly for expert purposes and an editor application for remote loading and storing of values to the SP and FF RAM. Figure 8 shows a screenshot of the initially developed synoptic application and Figure 9 shows a screenshot of the editor application. The

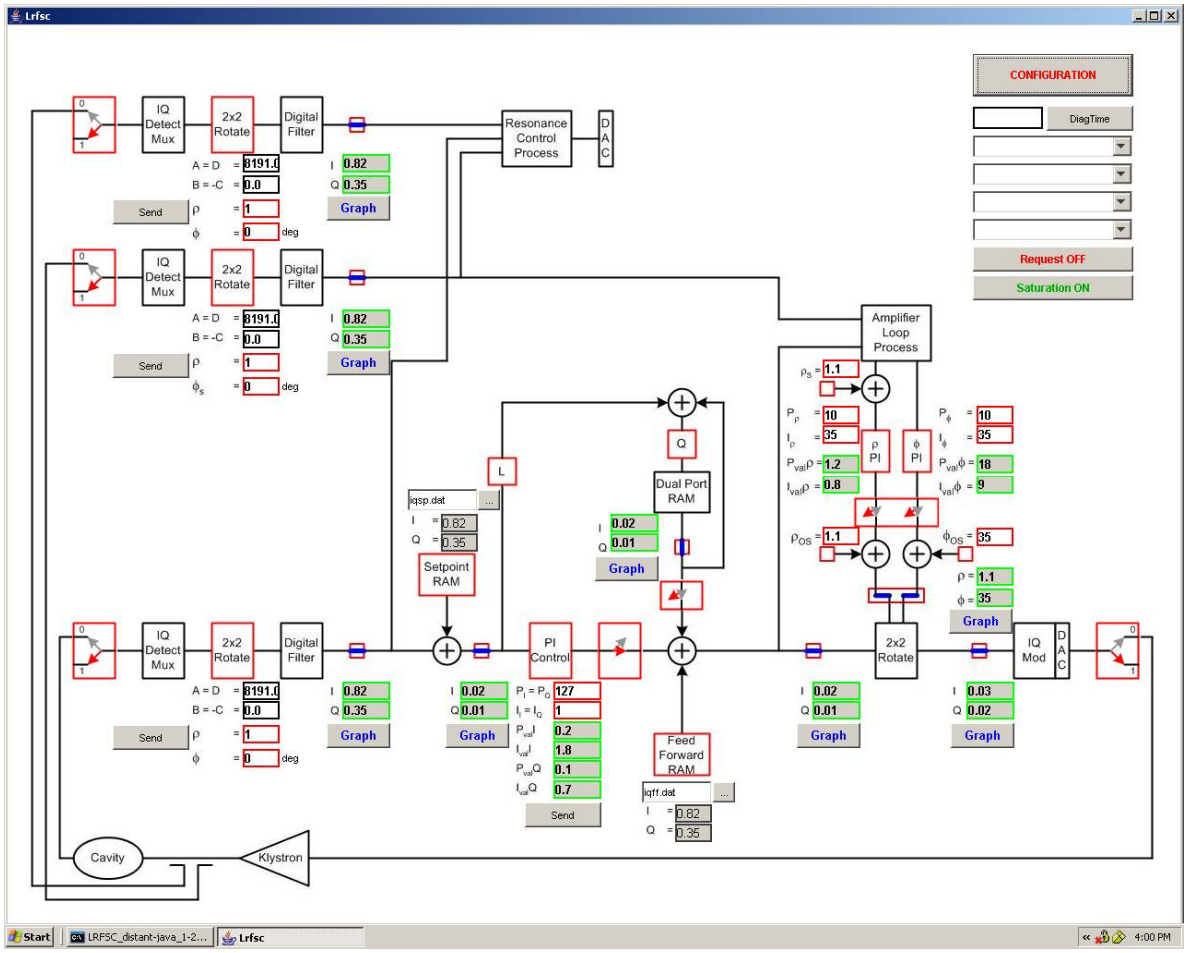

Figure 8. Screenshot of the Initial LRFSC Expert Application Developed by the FD

screenshots were provided by the FD and the SL to serve as basis for the new development.

The synoptic application provided a schematic view of the signal-processing loop controlled by the LRFSC system (see Figure 8). When starting this application, an expert user could connect to any running LRFSC device and adjust remotely its control loop parameters and check I/Q values captured at a given time at specific points in the control loop during the last processed cycle (the so-called "snapshots"). Data entry boxes on the synoptic view allowed the adjustment of parameters of the control loop. Through drop-down boxes (upper right of Figure 8), the source of the four diagnostic channels could be selected.

The editor application served for loading, storing, and graphically previewing of a list of I and Q values for the LLCC RAMs. The generation of this list had to be done using manually created spreadsheets or through the interface inside the application. When using the editor's I/Q values generation (the value entry boxes in the upper part of Figure 9), only simple ramps of amplitude and phase could be generated. When hitting the send button, the generated values were sent to either the SP or the FF memory. For reading of the values stored currently in the SP or FF memories, the application had to be reconnected to the specific frontend computer. After reading, the memory contents were previewed graphically in the chart in the lower half of the editor.

This initial software development for the LRFSC system conducted by the FD and the SL had to make the new hardware usable. Hence, only those capabilities of the hardware were

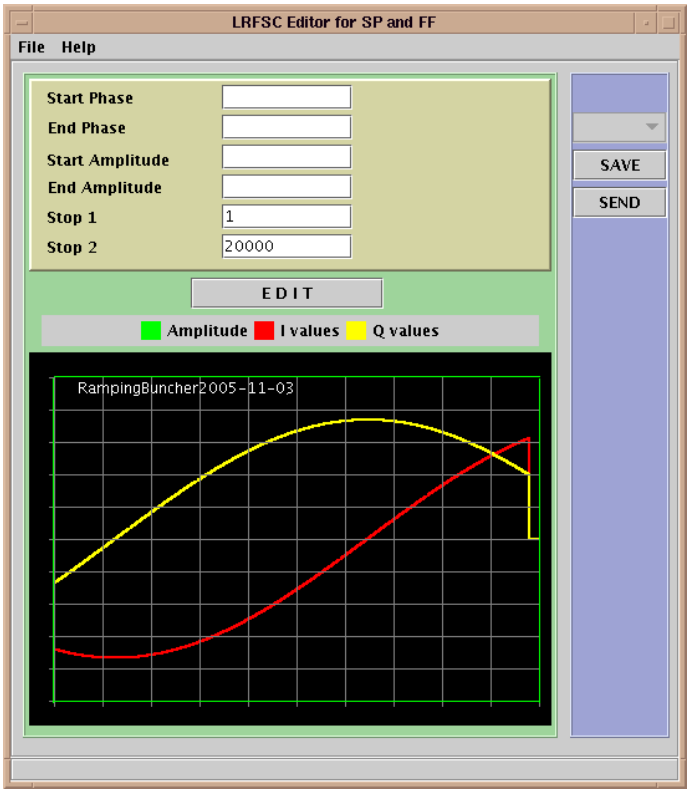

Figure 9. Screenshot of the Initial SP & FF Editor Application Developed by the FD

supported by the software that were needed for employing the system in the beam production of LINAC3. Initially intended features such as the cavity resonance control output were not implemented. The server side software for the frontend system as well as the client side were developed together and thus had strong dependencies between each other. For example, the FESA class had to be aware of the implementation of the LLCC driver to properly hand over data fields. All layers of the software had been developed from scratch by the FD and the SL.

The focus of this initial software development was to make things work. The design was oriented towards primary functionality. Modularity and information hiding (see [79]) were not used. In addition, the Java Graphical User Interface (GUI) applications did not use object-oriented concepts of the programming language. This all resulted in a code base characterized by duplicated code fragments instead of modular code and code reuse (for example between the two GUI applications). Moreover, besides a memory map for the LLCC interface, no documentation has been produced along with the initial development describing the architecture or design decisions. In addition, in-code documentation has been produced only sparsely. In order to figure out what a specific element of the client application really did, it had to be traced down through the layers of software down to the FPGA. With some interactive elements of the synoptic application, this showed that features shown to the user were not implemented on all software layers. For example, the part under the “amplifier loop process” in Figure 8 showed interactive elements to the user, but those features were not implemented in the FPGA. This means that the LLCC did not have these signal processing elements.

One relevant design decision had been retraced by the MD during the analysis of the former implementation, namely the overall handling of the I/Q values for the SP and FF memories. As the LLCC works in a pulsed mode, where the same signal processing is repeated for every beam bunch, the I/Q memories of the LLCC were dimensioned to hold a pair of I and Q values for every single digital time step of the processing of a single pulse. Based on the maximum pulse duration, 20,000 I/Q value pairs had to be stored in the RAM banks. Even though the editor application allowed only the setting of simple value ramps consisting of a single rising linear segment, all 20,000 pairs were calculated by the client application and then sent to the FESA class through the middleware and from there on handed down to the driver of the LLCC. Along with the pairs, also the values to generate it were sent to the FESA class so that they could be reconstructed at a later time or when another client connected to the system. Since also manually generated value pairs could be sent to the device, inconsistencies in the system could arise. The overall software was not designed to handle more than one client simultaneously. However, this could happen since the operators of the system are located in a control room on a different site than the expert users. Related to this, an issue with the software was that the user had no visual confirmation through the GUI that the parameter values he is seeing are those that actually are set inside the LLCC's RAM and FPGA.

This description of the initially developed software package for the LRFSC system shows that it had been developed with primary functionality in mind. In this way, it served several years in the beam production of LINAC3. As explained earlier, organizational changes as well as

technical changes in the middleware triggered the redevelopment effort. This redevelopment focused not only on the necessary changes but also on tackling the described flaws and on new requirements that only emerged after the initial development had been finished.

Unfortunately, no parts of the synoptic client application could be reused for the redevelopment effort. The program code implementing the graphical elements was directly coded into a monolithic class defining the whole user interface. The new requirements thus affected the whole user interface implementation. Moreover, the middleware communication was not fully decoupled from the user interface by a separate software layer. Thus, as the middleware changed, the former implementation became obsolete. Furthermore, also the editor application code could not be reused. Its rudimentary design and the new requirements affecting the way of editing the I/Q value ramps made that previous code obsolete as well.

3.2 The Project's Outcome

This section describes the outcome of the work executed during the redevelopment project. It presents the new client applications developed by the main developer along with specific elaborated solutions for the overall software package. The differences to the former software solution are highlighted. Furthermore, new design decisions are explained and some technical details are given about the new software solutions.

3.2.1 The New Developed Software Package and Elaborated Solutions

During the project under study, a complete software package has been developed for the LRFSC system. This included:

- three new Java client GUI applications sharing a common device abstraction model,
- a new FESA class definition with a custom code implementation,
- a new driver for the LLCC with an additional driver API library, and
- a modified version the VHDL code defining the logic of the LLCC's FPGA[3].

The biggest part of the new software package, the new Java client GUI applications[4] as well as the FESA class definition and custom code implementation were designed and implemented by the main developer (MD) (who is also the author of this study). The high-level software expert (HE) contributed to the middleware part of the client software design. The FESA class design was also done by the main developer and its implementation has been developed in collaboration with the former developer (FD). The LLCC driver and its API library were developed by the low-level software expert (LE) and the VHDL code of the FPGA have been reworked by the section leader (SL). For the driver, its API and the FPGA code, the MD

[3] This was necessary to incorporate new features and a radical change in handling the I/Q value pairs by the LLCC. These sequences of I and Q value pairs modulating the amplitude and the phase of the RF field were called "waveforms" in the context of this redevelopment, as their exact nature (i.e., their implementation) was not clear initially.

[4] The three Java client applications developed by the MD consist in sum of over 17,000 lines of code.

collaborated in their design and for the interface definitions. The MD's duty was further to make the new LRFSC software conceptually as coherent as possible.

The following sections describe the new software package on the client side and on the server side. On the client side, the focus is on each application individually but also the common solutions shared between them are highlighted. Based on these descriptions, the flaws of the previous solution are addressed together with solutions provided by the new software.

3.2.1.1 New Requirements

During the course of this redevelopment project, new requirements were identified that had an impact on the overall software package. The following list summarizes the most relevant requirements represented in form of a "feature list"[5]:

- The software package shall be "cleanly" implemented for handover to the group taking over the responsibility.
- The system shall support the PPM/Multiplexing mode (i.e. the change of configuration for each cycle).
- The waveforms shall be automatically generated "on the fly" by the system based on linear interpolation between "breakpoints" defining fixed I/Q pairs at given time steps.
- The breakpoints shall be editable graphically and as a table.
- A repeat mode shall be provided to generate repeating waveforms from a single sequence of breakpoints.
- The "expert only" features shall be separated from those available to the operators.
- All available features shall be controllable directly from the expert application.
- A remote oscilloscope function shall visualize the diagnostic channels.
- The diagnostics features of the system shall allow the selection of a given RF pulse within a cycle.
- An interface to a resonance control system shall be provided.
- The system shall be in a "safe mode" after startup.
- The GUI applications shall display values in different formats (raw decimal I/Q, normalized I/Q, amplitude/phase).
- The GUI application shall inform the user if the values displayed correspond to the values currently set in the frontend and provide a way to refresh the view.
- The software shall only use libraries supported by CERN for controls software and shall use the latest CERN controls middleware CMW.
- The client application(s) shall be compatible with the software infrastructure in the CERN Control Center (CCC) (e.g. Java Webstart via JNLP, context from environment, message console).
- The GUI client application should be compatible as far as possible with the previously implemented frontend computer software.

[5] This list of textual representations of the requirements is simplified in order to give the reader a "sketchy" overview. During the project, the corresponding requirements were captured in diverse form (see also 4.3).The following keywords are in the list: "shall" for mandatory requirements and "should" for optional requirements.

- The CERN "wheelswitches" (see application screenshots further down) should be employed by the GUI applications where possible, as operators prefer mouse-only control.
- The former strict distinction between a production and a configuration mode should be removed.

These requirements lead to changes on all parts of the LRFSC software. On the client side, the resulting design consisted of three different GUI applications. Several solutions have been elaborated by the MD that are shared by the three applications.

3.2.1.2 The Expert Client Application

The expert application named LRFSC-XP (screenshot in Figure 10) was developed by the MD. It replaced the previous synoptic expert application. Similar to the previous synoptic application, it represents the signal-processing loop controlling the RF field of the cavity (which can be found on the right border in Figure 10). The new layout features only those elements in the loop that are really existing (i.e., only the implemented features) and provides additional information to the expert user by showing the LLCC front connectors as part of the

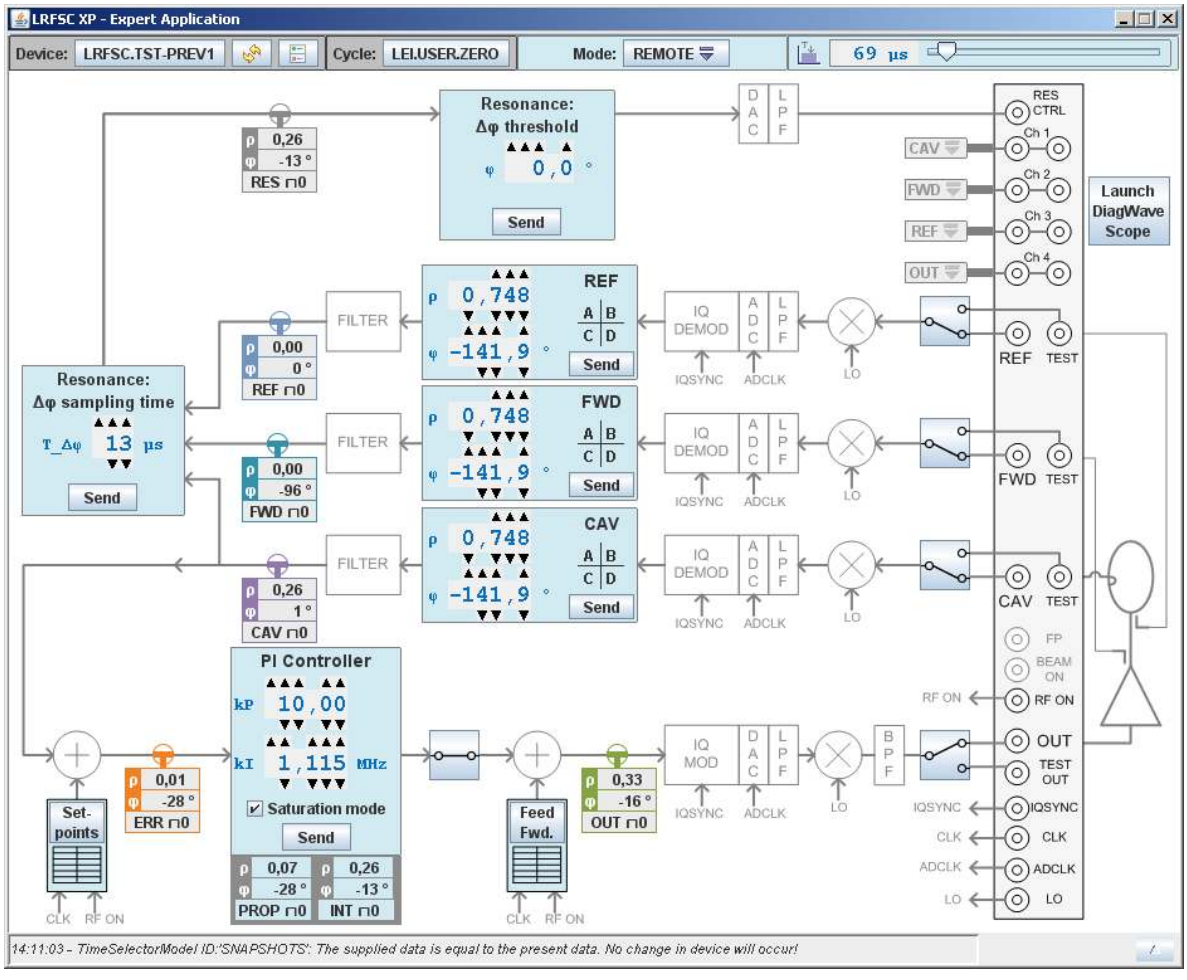

Figure 10. Screenshot of the New LRFSC-XP Expert Application (Developed by the MD)

loop (see right side of Figure 10).

This new layout emphasizes the fact that the control loop is closed through connecting the machine on LLCC connectors. Signals coming from the cavity through the connectors (CAV, REF, and FWD connectors) on the right are preprocessed in the upper half of the loop, then manipulated by the SP waveform, the PI controller, and the FF waveform and finally converted to the OUT signal driving the RF field (through the OUT connector and the amplifier). In comparison to the previous solution, the new application introduces a more intuitive scheme of GUI interactions: Firstly, only colored elements are interactive. Gray elements either are static parts of the control loop (e.g., DAC, LPF) or deactivated in the current mode (as the diagnostic source switches in the upper right part in Figure 10).

Moreover, functional blocks are visually delimited by forming blocks. In this way, it is visible which elements belong to which block. In the main view of the application, five different types of such blocks are visible: blocks with numeric inputs (REF, FWD, and CAV rotation matrix controls, PI controller, two resonance control blocks), signal switch blocks (front panel connecter switches, loop close switch), selector blocks (mode, four diagnostic channel source selectors), diagnostic "snapshot" blocks (RES, REF, FWD, CAV, etc. in different colors), and finally a time slider block (snapshot time selector). A special type of blocks are those with a button. They allow the expert user to automatically open the two other client applications. The mentioned snapshot blocks are symbolized as signal pickup points along the processing loop. They show acquired values at these points at the time of a pulse selected by the time slider visible on top of the screenshot. They are intended for easy fine-tuning of the control loop settings. Their values update automatically each time a given pulse occurs during the selected cycle. The snapshot blocks have different colors for easy identification, as some of these points in the loop can be selected as sources for acquisition by the diagnostic channels (see diagnostic scope application).

What is also visible in the screenshot in Figure 10 is that the CERN wheelswitches[6] have been used for all numeric input fields. These wheelswitches were, as all other fields, used in a way to allow the change of the value representations (e.g., amplitude/phase instead of I/Q). The overall application was designed to be fully controllable only by mouse input without the need of a keyboard. This includes the one-click launch of the waveform editor and the diagnostic scope.

The functional grouping into control blocks is one of the solutions elaborated by the MD for all three applications. As can be seen in the screenshot, each block (e.g., the CAV rotation matrix in the center) is delimited by a gray border. As soon as the LRFSC software detects that the values shown to the user in a given control block are no longer guaranteed to be the same as those present in the connected device, the block's border will change to a signaling yellow

[6] The “wheelswitches” are the interactive value input fields as, for example, in the upper part of the “PI Controller” in Figure 10. These wheelswitches are provided by a custom CERN Java library and are the preferred input elements for accelerator control application. They automatically maintain a predefined correct value range. Moreover, they allow the machine operators to use the client control applications on the large control screens with mouse-only input (i.e., without using a keyboard).

color. Depending on the case, this might affect a single block or all blocks (e.g., in case of a middleware connection problem). Additional information will be shown in the status bar on the bottom of the application (see lower border of the screenshot in Figure 10). A typical situation for this is that the user starts to modify control values such as the PI controller coefficients. This will make the "PI Controller" block signal that local values differ from the remote values. The yellow border would appear. As soon as the user hits the send button of the PI block, the new values will be transmitted to the connected device. A change of the block's border to normal would then signal the operation's success. By this principle, a block only shows up as normal if the remote values are equal to the locally displayed ones. This can be very useful in the case another user from a completely different location with access to the CERN technical network changes settings during the same time. Such an operation would result in inconsistencies for at least one of the users between the locally displayed values and the remotely set ones. The visual signalization would indicate the user that the settings he sent differ from those set in the device.

There are more reasons that can provoke the same result as described, such as middleware issues or problems with the timing system. What is relevant for the user to know is whether the values he sees are those really set in the connected device. In case such problems persist, the regular users of the LRFSC applications would then contact the persons responsible for the system in question for further investigation.

Finally, in the top left corner of the application (as well as for the other two applications) are visible the common controls of the LRFSC applications that were elaborated by the MD. Those allow selecting the LRFSC device to connect with and the cycle for which the configuration has to be modified. As only few parameters of the expert application are cycle-dependent, the cycle selection is more relevant in the other two client applications. In addition, the standard CERN message bar that is visible on the bottom of all the three application screenshots was introduced as a common element. These are the common interactive elements that all three application share.

3.2.1.3 The Wave Editor Client Application

The wave editor application named "LRFSC Waveform Editor" (screenshot in Figure 11) replaced the previous "SP and FF editor" application. As the methods for generating the I/Q value pairs for Set-Points and Feed-Forward that are necessary in the signal processing control loop have been radically modified, also the client application for working with these parameters completely changed. The waveform editor is the primary application to be used by the machine operators in the CERN Control Room for controlling the LRFSC systems. Therefore, effort has been put by the MD into the development of a full graphical editor for the waveforms with several specific features.

In the new application, a graphical user interface allows the user to visually modify the so-called breakpoints that define the Set-Points and Feed-Forward waveform (which waveform is edited can be selected). The visual manipulations include the insertion, displacement, and

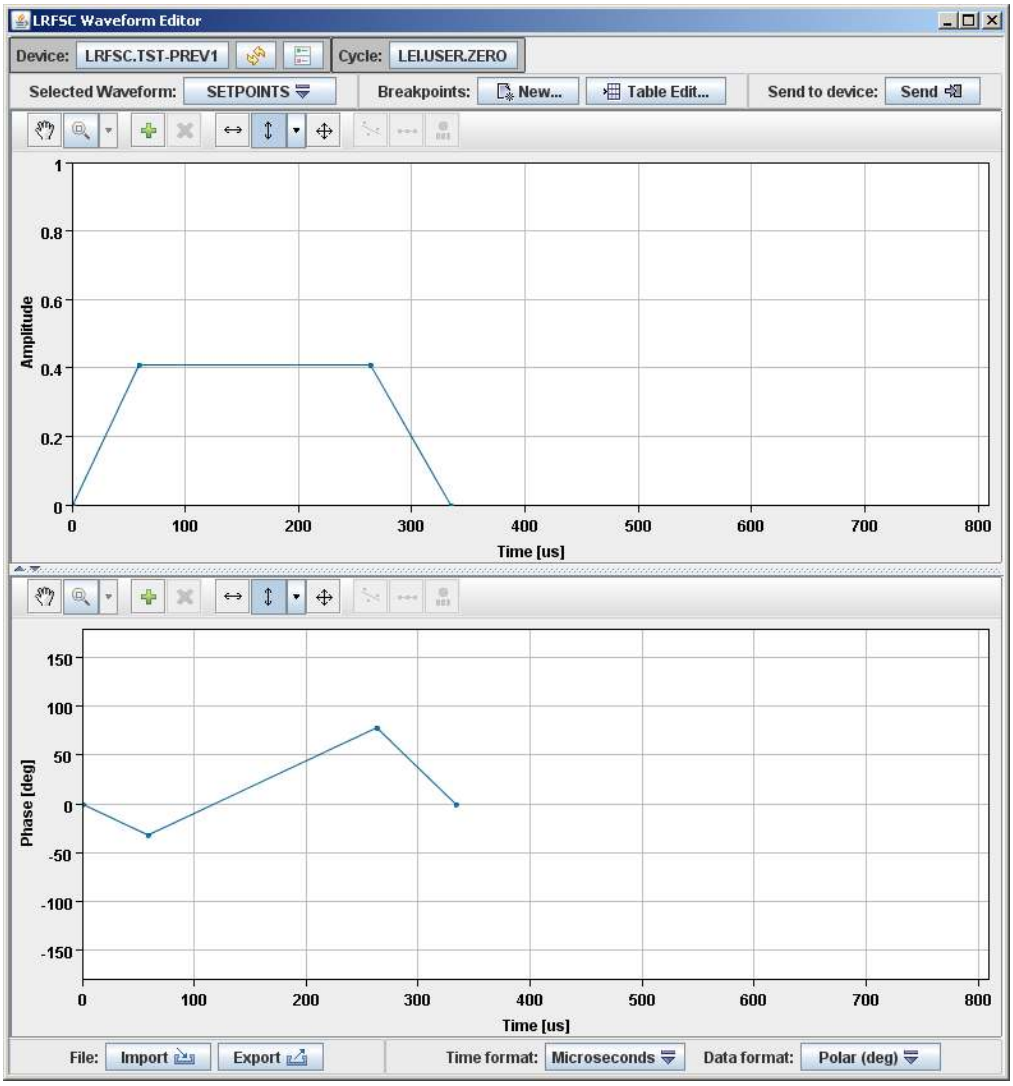

Figure 11. Screenshot of the New LRFSC Waveform Editor Application (Developed by the MD)

removal of single breakpoints. Furthermore, group operations such as adding an offset can be applied to a group of selected breakpoints.

As can be seen in the screenshot in Figure 11, the main interface is divided into two graphs showing the breakpoints. As the waveforms are used by the signal processing logic to modulate the I/Q data stream, they are also composed of I/Q value pairs. Since I/Q pairs (the rectangular format) can be converted into amplitude/phase value pairs (the polar format), two different representations are possible. The representation format can be selected in the new application (see lower right corner in the screenshot). In both formats, there are always two separate dimensions I and Q or amplitude and phase. Therefore, the main user interface is divided into two graphs showing both dimensions of the selected format. The breakpoints can be seen in the screenshot as dots in the amplitude (upper part) and in the phase (lower part). Between these breakpoints, the waveform is interpolated linearly by the system.

The most important feature of this editor is that the edited set is automatically maintained consistent with respect to the constraints imposed by the LRFSC frontend system. These constraints are, for example, respecting the minimal and maximal number of breakpoints, keeping the first point at time step 0, or respecting the minimal time step difference between two adjacent points. Keeping the consistency also includes maintaining the edited points as a pair of an I and Q value or amplitude and phase for a certain point on the time axis. Modifying the placement of one point on the time axis automatically adjusts the whole pair on the time axis. In addition, the insertion and deletion of points modifies a whole pair. Moreover, a breakpoint data set received from the connected frontend system is also checked for consistency in order to detect if the set was corrupted by direct access on the frontend system.

An additional feature is that the breakpoints can be edited in a precise manner using the "Table Edit" feature (button above the graphs). This allows one to open a separate window showing the breakpoints as a table of time steps and the two value dimensions (see screenshot in Figure 12). By clicking into a given line, wheelswitches to edit the values show up. The changes made in the table are reflected by the graphs in the main interface. This allows a precise editing of the breakpoints. By using the wheelswitches the table view can be controlled by using only mouse inputs without a keyboard (requirement for control applications, see 3.2.1.1). Data consistency is automatically enforced by the wheelswitches as their minimum and maximum values are set so that the set of breakpoints maintains its consistency as described above.

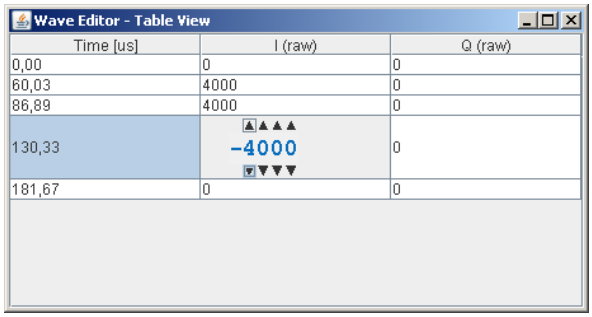

Figure 12. Screenshot of the Table Edit Feature of the New Waveform Editor Application

Finally, import and export of a set of breakpoints to and from local files on the user's client computer is also possible. The consistency check described above is also necessary for importing settings from a file in order to avoid undesired behavior by manually edited breakpoint sets. After editing, a click on the send button sends the waveforms to the currently connected frontend system. When connecting to an LRFSC device, the breakpoint set currently stored in the frontend is retrieved. Similar to the expert application, inconsistency between the data currently displayed and the data stored in the connected device is highlighted to the user by a yellow signalization border around the control block. The only control block in this application is the graphical editor. Nevertheless, it shares that same concept. In addition, the user is warned by a message if the waveform is changed on the device during local editing. Thus, the user is warned about simultaneous access from other users.

3.2.1.4 The Diagnostics Client Application

The remote oscilloscope application for the diagnostic channels named "LRFSC Diagnostic Scope" (screenshot in Figure 13) replaced the previous "graph" function of the synoptic expert application. It was also fully developed by the MD. As with the waveform editor described above, the I/Q data of the signal processing loop can be represented in different formats that always consist of value pairs along a time axis. Hence, also this application's interface is composed of two graphs showing the two value dimensions (I in the upper half and Q in the lower half in the screenshot).

The application allows the selection of I/Q data streams from different predefined sources inside the digital signal processing loop for acquisition by each of the four diagnostic channels (selectors in the top right part of Figure 13). Those signals are then acquired by the LLCC during the processing of beam pulses. That data for a given pulse can then be requested by this diagnostic application, which displays it as waveforms. During beam production, new data is gathered by the application each time the selected pulse for a chosen cycle occurs. In this

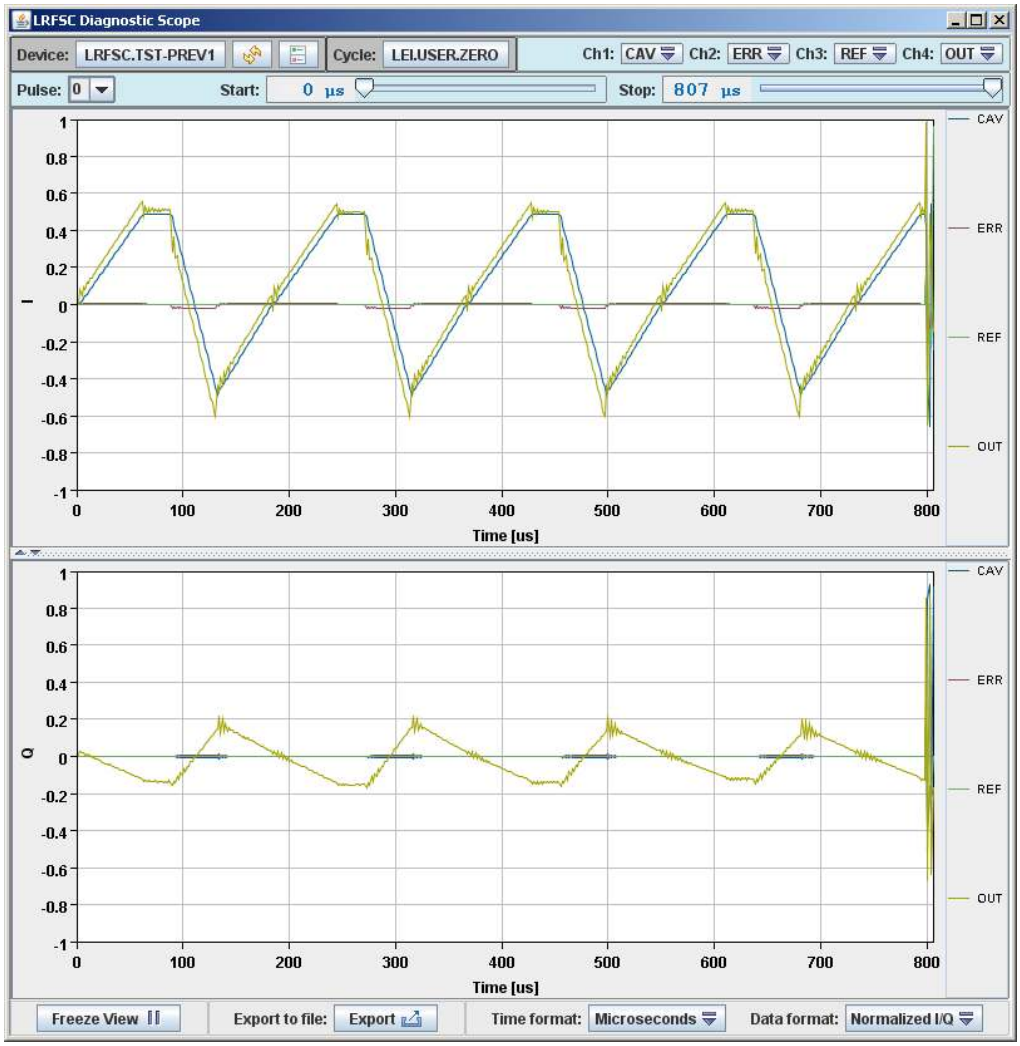

Figure 13. Screenshot of the New LRFSC Diagnostic Scope Application (Developed by the MD)

case, the view is refreshed automatically unless the user chooses to freeze the view (button in lower left corner of screenshot). These features make the application usable like a remote digital storage oscilloscope for the selected signals of the control loop.

A very important new feature of this remote scope is that it allows zooming through "reframing": When the user uses the graphical zoom tool on the waveform displays, the start- and end-time of the selected zoom window is sent to the frontend system requesting it to acquire the signals in the specified time window. This allows the user to see the full evolution of the signals over time as well as detailed down to the maximum possible time resolution. This time window is also selectable by numeric inputs and sliders on top of the waveform graphs (see Figure 13). As the possible time windows are limited by implementation constraints, the server-side software adjusts automatically to the closest possible time window.

As with the other two GUI applications, control blocks signal if their current values are inconsistent with those in the device. This is important for the signal source selectors, the pulse selector, as well as the time windows sliders. Also the scope view follows this general concept: If the user freezes the view (see button in the lower left corner in Figure 13), the border of the scope will signal inconsistency as soon as the new data has been acquired remotely by the LLCC making the current view outdated. By the signalization through the border color only, the user is informed without interrupting his current work with the application.

Finally, it should be noted that an export feature was integrated by the MD to allow saving the displayed waveforms to a local file on the user's client for later analysis by external programs.

3.2.1.5 Server-side Software on the FEC and New Operation Principles

On the server side (i.e., on the frontend computer), the new requirements necessitated a complete redevelopment of a new FESA class as well as a new LLCC driver with an additional API library. Furthermore, changes to the FPGA logic became necessary. The reimplementation of the server side was based on the newly designed general "operation principles" of the LRFSC system that originated from the limitations of the given LLCC hardware. These new principles, that also had an impact on the client side, are described in the following.

Breakpoint-Based RF Field Control

CERN's PPM/multiplexing scheme and the awareness of the system for LINAC3 pulses were fully introduced in the new LRFSC software solution. Due to technical limitations imposed by the hardware, the new breakpoints principle had to be elaborated and implemented on the server side. This principle consisted of moving from waveforms for the modulation of the RF field that were generated as a sequence of 20,000 points by the previous client application to the generation of these waveforms "on the fly" out of "breakpoints" during the processing of a pulse directly by the LLCC.

Waveforms are now formed by the card's FPGA using linear interpolation between each sequence of adjacent breakpoints in time. This allows the LRFSC system to perform even very complex amplitude and phase ramps of the RF field for each cycle individually. Additionally, a repeat mode was introduced to generate high frequency repeating waveforms for calibration and testing purposes. As the modulation of the RF field through the SP and FF waveforms in conjunction with the digital PI controller is the most important feature of the LRFSC system, this new principle of operation had an impact on all software layers.

Distinction of Remote and Local Diagnostic Modes

The other important features of the LRFSC system are its integrated diagnostic capabilities. These consist of the snapshot points shown to the user in the expert application and the diagnostic channels shown in the diagnostic scope application. The snapshots acquire single I/Q value pairs at certain points in the control loop at a specified time step measured from the beginning of a pulse. At some of these points in the loop, the LLCC can also redirect the occurring value stream during processing to memories acquiring this stream. The memories can then be read out remotely. On the same time, these streams in form of I/Q value pairs can be converted to analog I and Q signals that are conducted to outputs at the LLCC's front panel for local diagnostics through a connected oscilloscope.

Due to the limitation of the LLCC hardware and the new awareness of the system for pulse numbers and cycles, the simultaneous use of the local and remote diagnostic channels had to be reconsidered. Therefore, a distinction between a “local” and a “remote” mode has been introduced into the server-side software. On the client side, only the expert application allows the setting of this mode, as only machine experts will use the diagnostic outputs of the card. Normal operators will use the remote scope function. If an expert uses the local mode, the remote diagnostic application shows a deactivated state. In local mode, the expert application controls the selection of the sources of the diagnostic channels for the LLCC outputs (see upper right part of Figure 10). In remote mode, those selectors are deactivated and instead the signal selection can be operated through the remote diagnostic scope application, which is available to the machine operators.

In addition, a requirement was to have a default “safe” mode in which an LRFSC system automatically starts up after power up. This mode replaced the previous configuration mode that became obsolete. The new mode was added to the other two operation modes into a single selectable property of the frontend system, as the modes are mutually exclusive. Since only the expert application allows the setting of the mode, after a power down of an LRFSC frontend, an expert user has to check the machine configuration through the expert application and switch to “remote” mode to allow operators to use the LRFSC system. In this way, the new principle of operation with these distinct modes helped to satisfy several different requirements (related to local and remote diagnostics, the configuration mode, and the “safe mode”).

Conceptual Consistency

Beside these principles on the server side, an important aspect of the redevelopment of the whole software package was to get a software package that is “conceptually consistent”.

From a user point of view, the new applications were designed by the MD to offer easy-to-use interactive elements that all work in a “similar way”. They indicate by the same intuitive principle if the value or state they are showing is also present in the device. The wheelswitches as preferred control elements are used wherever applicable. If visual sliders are used, an additional numeric entry field that works in the same way as the entry field of the wheelswitches is provided. Notations are used consistently throughout the GUI applications. A goal of the newly designed applications was to give the user the feeling that all the applications “belong” to the same system by their “look and feel”.

From a developer point of view, consistency was enforced by using the same functional grouping along the different software layers as far as possible. A functional block of the LLCC's control loop forms a single property with all necessary fields in the FESA class, which further corresponds to an interactive block in the GUI code. Redundancy that was present in the previous software solution was eliminated. In the new solution, interfaces are defined between the different layers of software using names of properties or data fields in a consistent way.

3.2.1.6 Solutions for Previous Flaws

Based on the description of the new LRFSC software package, it is now possible to compare it to the initial software solution and to address the flaws mentioned in Section 3.1.4. The following list addresses them:

- In the initial implementation by the FD, the FESA class code made direct calls to the LLCC driver and defined thus no “clean” interface. In the new implementation, the FESA class uses a driver API library to access functionality of the LLCC. This library then calls the driver itself. The driver API library was developed to define such a “clean” interface. The previous implementation also lacked a properly defined interface between the GUI application and the client side middleware. In the new implementation made by the MD, an additional layer was introduced on the client side that decouples the client middleware from the GUI. Modifications in the middleware now do not affect the GUI applications directly. Using this decoupling, it was even possible to provide a compatibility layer for older versions of the LRFSC FESA class.
- The initial software package showed an inconsistent set of features between what the GUI offered to the user and what the system was capable of doing. These inconsistencies were avoided in the new implementation.
- The previous implementation was developed using a code-and-fix strategy. Especially the GUI Java code relied on duplicated code segments. External changes, such as those affecting the selection of available devices, necessitated code adaptations at several places. The new implementation follows an design decoupling the user interface from

middleware library calls and avoids duplicated code. Changes, e.g., on the device selector, would affect only small parts of the code.

- In the initial GUI applications, the concrete LRFSC device to work with could only be chosen at the start of the application. Unfortunately, the former SP and FF editor also loaded the waveforms from the device only on startup from a selected memory. The new applications provide a commonly shared selector for the LRFSC device to connect to and the cycle to work with, thus avoiding a restart for a change from SP to FF waveforms or between different devices.
- Potential inconsistencies between the values displayed and those stored in the device could remain unnoticed to the user in the previous implementation. Especially simultaneous interaction with the frontend system from two different clients could occur unnoticed and result in unintended machine behavior. Using the new principle of inconsistency-signaling developed by the MD and an automatic subscription for change notifications make the remote control of LRFSC frontend systems less error prone.
- A special case of inconsistency was possible in the previous implementation while dealing with an amplitude and phase ramp to generate the SP and FF waveforms. The values defining the simple ramp of waveforms were retrieved upon start of the old SP and FF editor separately from the waveforms. The waveform then displayed to the user could be one that did not correspond to the ramp values. The new implementation introduced the operation principle of breakpoints with on-the-fly linear interpolation in the FPGA. Through this, sets of breakpoints are transmitted between the frontend system and the waveform editor and are displayed as such. Only during signal processing operation, the LLCC generates the necessary waveforms. This keeps the remote view consistent with the data in the frontend system. The previous possible inconsistency through data redundancy was eliminated.
- Finally, the former “SP and FF editor” offered only a rudimentary feature set for waveform generation. The new implementation solved this flaw by providing a feature rich visual editor for the new breakpoint principle that defines the SP and FF waveforms.

The subsequent section shall give a few technical details about the most relevant solutions described above.

3.2.2 Technical Specificities of the Elaborated Solutions

The complete redevelopment of the LRFSC software package led to changes in the software layer structure. The new structure is shown in Figure 14. It is discussed along with technical descriptions of some of the elaborated solutions.

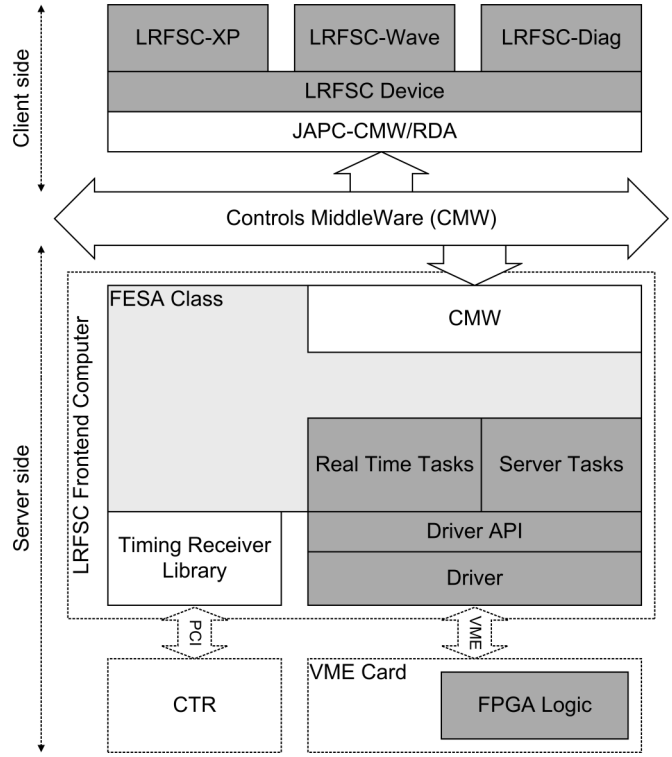

Figure 14. Software Layers of the New LRFSC Implementation

When compared to the layers of the initially developed solution (see Figure 7), two major differences appear. One layer has been added between the driver (communicating over the VME bus with the LLCC) and the FESA class and one additional layer is inserted between the client side CMW and the GUI applications. In addition, the CMW client has changed from the initial CMW device API implementation in Java to the "JAPC-CMW/RDA" library.

3.2.2.1 Client side

On the client side, the layers structure depicted in Figure 14 shows that the three GUI applications are not completely independent. Since all three GUI applications have to communicate with LRFSC systems using the CERN middleware, they have common parts in their design. The common parts related to the middleware communication with the device have been encapsulated the additional "LRFSC Device" layer that hides the middleware access for the GUI applications. The design elaborated by the MD can be represented in a simplified way as in the class diagram in Figure 15.

LRFSC GUI applications can connect to running LRFSC FESA class instances through the Device layer. This can be seen in the lower left part of the UML class diagram of the client-side software shown in Figure 15, where the "Device" layer is represented. As the layer provides a higher-level interface[7] to a GUI application[8] than the underlying JAPC, it was possible to create

[7] The interface is named "DeviceInterface" in the design as shown in Figure 15.
[8] All three GUI applications inherit from the "LrfscAbstractApp", which connects to any implementation of the "AbstractDevice" through the "DeviceInterface".

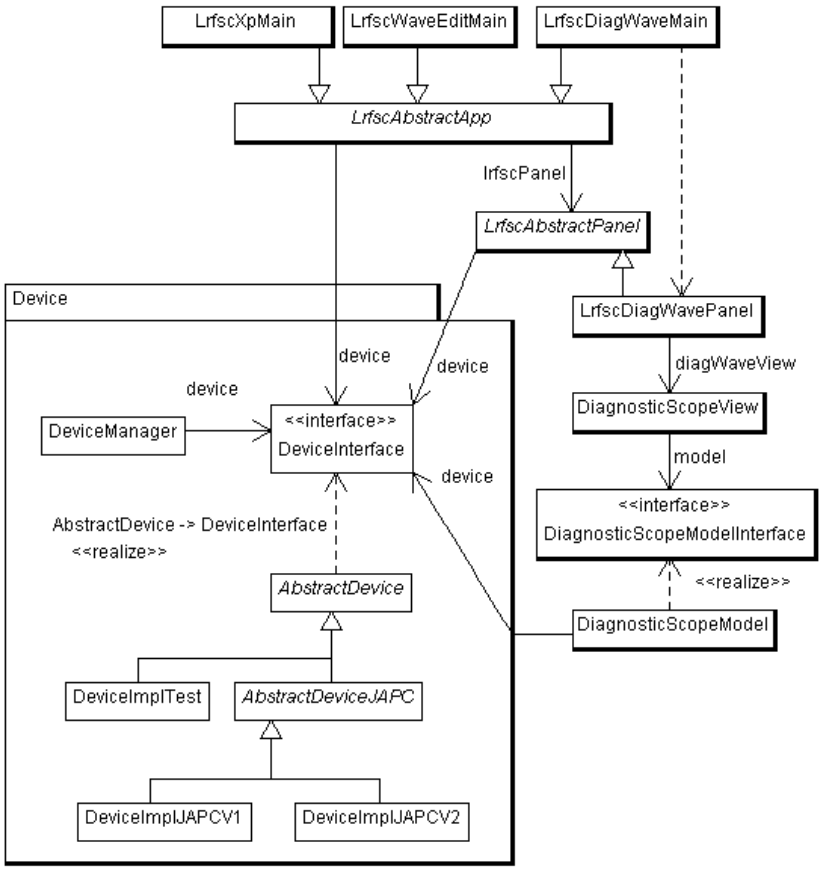

Figure 15. UML Class Diagram of the New LRFSC Client-side Software Developed by the MD

different implementations of this layer. One version was created to communicate with the new LRFSC FESA class[9] (based on FESA 2.10), one for the initially implemented FESA class[10] (based on FESA 2.9), and one as local simulation of a running LRFSC FESA class instance without using CMW communication at all. A startup parameter decides which one of the implementations of this layer is instantiated.

In the case of the local test-device simulation, the "DeviceImplTest" class also instantiates a separate GUI window that allows the developer to virtually generate changes in the Device layer. The effect of these changes on the GUI applications can thus be tested systematically. In the same way, changes of values made in the GUI applications and sent to the test-device can be checked by the local simulation.

In the case of the "real" Device classes that communicate through CMW with a FESA class on the server side, a JAPC subscription for all device properties is made. New property data, therefore arrives through the JAPC-EXT-CMW library at each occurrence of the cycle that is

[9] An LRFSC frontend system running this version of the FESA class is accessed through the "DeviceImplJAPCV2" implementation of the Device layer.

[10] An LRFSC frontend system still running the older version of the FESA class can also be used. The "DeviceImplJAPCV1" implementation provides the compatibility in a transparent way to the GUI applications.

currently selected or when data changes on the server side. The Device layer stores locally all property data each time new data is read from the server. In this way, the local data set always reflects the remote data as long as the subscription is active.

When property data is read that differs from the locally stored data, a change event is fired to all listeners subscribed to the Device layer. Using getter and setter methods, the listeners can then request the current local data. This principle is the basis for the inconsistency signalization of GUI control blocks as described above. If a user starts to modify data in his local view, the values he sees differ from those stored in the device. Depending on whether the GUI block is an automatic sender or a manual send button is provided, the local view will signal inconsistency as long as the Device layer does not signal that it has the locally latest data of the remote device. In this way, the GUI block can be refreshed and signal a normal consistent state to the user. Additionally, the global refresh function provided by the GUI applications (double-arrow button on top of the screenshots of the three new applications) also uses this principle. When clicked by the user, all control blocks in the view will receive an inconsistency notification. They will show an inconsistent state as long as they do not receive data update notification from the Device layer and gather the currently valid values.

The GUI control blocks of the user interface of the client applications are based on a variation of the “Model View Controller” (MVC) pattern that combines the view and the controller class, similar to Java’s Swing “Model Delegate” pattern [72 p. 15]. Moreover, the chosen design features, similarly to the “Model View Presenter” (MVP) pattern, an additional interface for the model class allowing it to be interchangeable. This constellation is shown on the example of the GUI block providing the remote scope functionality in the simplified class diagram. The model class (“DiagnosticScopeModel”) holds the data shown to the user by a view class (“DiagnosticScopeView”) that is registered as an observer of the model class according to the “Observer” pattern. This makes several alternative views for the same data model possible. Through the model interface (“DiagnosticScopeModelInterface”), the graphic views can be reused when the model changes.

Each model type roughly represents a certain type of CMW property defined in the FESA class on the server side. Hence, the local GUI blocks represent properties of the LRFSC frontend as functional groups. The controllers of the MVC design register as change-listeners at the Device layer following an object-oriented “Observer” pattern. The views are usually graphical objects (derived from a Java Swing class) and they are placed within the main panels of the LRFSC GUI applications as needed. From a GUI point of view, the applications' user interfaces are simply built up by placing the desired views as local control blocks of the remote properties at the desired places in the main user interface panels. The functionality of the blocks is then automatically handled through the described design. In the example of the GUI block providing the remote scope functionality, the “DiagnosticScopeView” is simply placed in the “LrfscDiagWavePanel”, which is the main user interface panel of the Diagnostic Scope application (see Figure 13).

In addition to a given set of GUI control blocks, all three LRFSC applications have common control elements that are not local views of remote properties. These control elements are the same in all three applications and consist mainly of the "Device Selector" and the "Cycle Selector" and a common application frame using the CERN status bar defined in an abstract class ("LrfscAbstractApp" in Figure 15). The three GUI applications extend this abstract LRFSC application class. The three LRFSC applications thus only differ by the control blocks placed in their GUI panels (the concrete implementations of the "LrfscAbstractPanel", e.g., the "LrfscDiagWavePanel"). This design ensures efficient sharing of the code necessary for all three applications. It allows furthermore the launch of a given application within another one but sharing the same cycle and device selectors and the same Device layer. This allows the expert application to integrate the other two applications without launching them as true separate applications.

An interesting special case in this design is the graphical wave editor block used as the principal control block in the wave editor application. This special control block is based on a graphical chart provided by a CERN Java library called JDataViewer (see [66]), which is based on a MVC design. In this case, the model holding a list of the breakpoints was the data holder class of the JDataViewer. By implementing custom extensions to the classes of the JDataViewer and by using two instances linked together, the data model could automatically satisfy the constraints imposed on the breakpoints set. Thus, the displaying of graphically editable points linked with linear lines was provided by that library. Another MVC implementation defining the overall breakpoints editor control block has been built so that it holds the two graphs (see screenshot of wave editor in Figure 11) and the surrounding special GUI controls. In this way, the additional functions such as the import and export were added. Moreover, a separate view was coupled with the data model of the graphs in order to provide the same data editable as a table.

3.2.2.2 Server side

On the server side, a new driver has been developed by the low-level software expert (LE) in collaboration with the MD. In addition to the driver, a driver API library was developed allowing the FESA class' actions to operate with LLCC data without using direct IOCTL system calls. This API encapsulates the concrete driver and memory map implementation and provides a higher-level access to the LLCC from the FESA class. The driver was implemented to support the newly added features and the new operation principles described previously following the added support for the PPM/multiplexed operation.

The biggest impact on the server side had the new solution of working with the SP and FF waveforms by generating them from breakpoints on the fly during the actual signal processing. By this principle, the I/Q value pairs previously coming from the SP and FF memories for every time step during the signal processing are now generated by a linear interpolator provided by the FPGA of the LLCC and its driver. This made changes in the VHDL code defining the FPGA logic necessary. Fortunately, the previous implementation could be

reused and the changes were implemented by the SL, who also developed the initial FPGA code.

For the waveform generation on the fly, the driver preprocesses the two (SP and FF) breakpoint sets for each cycle and stores the resulting data sequences at predefined places in the RAM memories of the LLCC that previously held the full waveforms. The preprocessing transforms each breakpoint (consisting of a triplet of time step, I value, and Q value) using the data of the next following breakpoint into a sequence of triplets consisting of a time step, an I value increment, and a Q value increment. Before the processing of an RF pulse begins, the FPGA is instructed which cycle is played and therefore, which preprocessed data set has to be used. During the signal processing, the waveform generator logic sequentially loads a preprocessed triplets each time the time step counter (counting from the start of a pulse) achieves the time step of the triplet. On every triplet loaded, the I and Q increments are loaded into an adder that adds the increments to an I and Q counter on each time step. This provides the linear interpolation that generates the SP and FF waveforms on the fly. Special predefined instructions in the sequence of triplets tell the waveform generator logic when a sequence of triplets ends and whether the sequence should be periodically repeated or the last value should be maintained until the end of the RF pulse.

The newly added PPM support had also an impact on the diagnostic channel function of the LLCC. For technical reasons, it implicated that only 512 value pairs per channel could be recorded and transmitted to the driver. Fortunately, this fitted well the approach of a remote scope, as displaying such waveforms on an average computer screen does not necessitate much more than this amount of data points. Nevertheless, it was desirable to be able to acquire a full overview of the waveforms as well as detailed views at the maximum possible resolution of one value pair per time step. This was achieved by implementing a time window for the signal acquisition directly into the LLCC's FPGA and its driver.

For the diagnostic acquisition, a start-time step as well as a skip-count were defined. During processing of a pulse, the LLCC waits for the start time step. When it occurs, the LLCC acquires the diagnostic channels every time the skip-count number of time steps elapses. It stops when the memory slot in each channel for the current pulse and cycle reaches the foreseen 512 value pairs. By choosing the maximum possible skip-count and 0 as start time step, the full signal evolution can be acquired. Pointing at a specific time step and using a skip-count of 1 allows the acquisition of the four selected signals at maximum possible time resolution.

In addition to this remote scope functionality developed by the SL, the LE, and the MD, the local outputs of the diagnostic channels on the LLCC front connectors had to be maintained. Since the RAM memories storing the signals fed to the diagnostic channels share the same data bus with the DACs producing the local diagnostic output (see Figure 6), a local and a remote diagnostic at the same time is not possible. The locally connected oscilloscope would see only the short timeframe specified by the driver in the FPGA (start time step and skip-count) instead of the signals during the whole pulse. Moreover, the readout of the stored signals through the VME bus at the end of the processing of each cycle would generate again

the acquired signals with false timing (readout clocked by the VME bus). Therefore, a separate remote and local diagnostic mode have been introduced. These modes were also implemented together with the new features into the FPGA logic. Using these modes together with the remote scope design described above, it was possible to satisfy all related requirements.

The introduction of PPM together with the handling of the separate RF pulses per cycle had also implications for the newly designed and implemented FESA class. The breakpoint principle required the design of device properties defining the sequence of breakpoints on the middleware side. Also the remote diagnostic scope function described above required newly defined properties in the FESA class, so that a middleware client can use the new features. Moreover, things such as the pulse selection for diagnostic snapshots or the mode selector had to be introduced. This new FESA class design as well as the necessary custom code implementations was elaborated by the MD in collaboration with the FD.

The reimplemented driver is now indirectly accessed from the FESA class through the driver API library, which offers a defined interface. Thus, the complexity of the custom parts of the FESA class task is greatly reduced, compared to the previous implementation. In its new form, the classes' main purpose is to provide the CMW server towards the higher control layer, to process remote requests for the manipulation of the LRFSC settings defined by the different properties, and to offer the middleware subscription mechanism for clients. This basic functionality is fully covered by the automatically generated code through the FESA framework. The development of the new FESA class consisted of defining the properties of the in the latest FESA shell tool along with all fields, data types, and options of the properties. In addition to the generated code, custom server actions were implemented to perform plausibility checks and some data format conversions on incoming settings for certain critical properties such as the breakpoints sets or the PI controller settings. Furthermore, the custom server action was necessary to send the incoming data after preprocessing towards the lower software layers. After plausibility checks and value conversion for some properties, the data is sent to the lower layer by calling the respective set-methods provided by the driver API. For reading data from the lower layer, a custom real-time action named “LrfscRT” was defined and implemented in the new FESA class, which is triggered by a timing event each cycle. When triggered, new data in the LLCC about the last cycle is available and can be requested using get-methods on the driver API library. For simplicity, all configuration and data is read back from the LLCC and stored in the data storage provided by FESA. From there, default server get-actions generated by FESA can be used through middleware calls to retrieve this data. Moreover, the FESA generated code takes care of notifying subscribed middleware clients that an event occurred and new data is available.

The handling of the right timing to access the LLCC through the VME bus was a collaborative task of the FESA class and the driver in the previous implementation. The whole system had to be switched into a special configuration mode to change settings in the LLCC and avoid interference with the signal processing. In the new implementation, the configuration mode became obsolete. By moving to a PPM operation, the system cannot be put anymore out of a production mode into a configuration mode during beam production. Continuous operation

has to be possible. Therefore, the complete communication with the LLCC is now encapsulated in the driver. The FESA class can call setter and getter methods of the driver API library at any time. The library then places IOCTL calls to the driver, which only manipulate a local buffer of the whole LLCC data (called the "module context"). Only when a specific VME interrupt occurs, through which the LLCC signals the end of an RF pulse, the driver can safely operate with the LLCC data through the VME bus until the beginning of the next RF pulse. Through this solution, all timing issues in accessing the LLCC have been hidden behind the driver API, which provides an interface to the LLCC at a higher level of abstraction.

Several conflicting requirements concerning the production and configuration modes of the system were raised during the course of this project and changed frequently, such as the "PPM operation" and the "20,000 points waveforms for SP & FF". Only through detailed knowledge about the LRFSC system and the LLCC hardware, these requirements conflicts could be resolved. Many such conflicting and changing requirements were elicited by the MD during the course of the project. They were eventually discarded as they became obsolete due to the better understanding of the "real needs" and the technical context. Along with this more detailed description of the most relevant elaborated solutions above, it emphasizes once again the crucial role that the specific domain knowledge played in the project.

4 RETROSPECTIVE ANALYSIS

This chapter presents the retrospective analysis that has been conducted by the author about the project described in Chapter 3. A case study approach as described in Chapter 2 has been pursued to analyze the evolution of the project and its influencing factors regarding the research questions of this study. This chapter first presents the data collected during the active part of the project and the data resulting from its analysis (as described in Chapter 2) and then elaborates on the three main topics in form of a retrospective point of view. Findings concerning communication issues, domain knowledge and requirements issues, and finally issues regarding the software development process are discussed. This structure reflects the common nature of the issues encountered during the project under study:

> "The major problems of our work are not so much technological as sociological in nature." [44 p. 4]

4.1 Data Presentation

As described in Section 2.3.3, the main data source in the present case consists of the diary notes that have been codified, crosschecked with other sources, and preprocessed for analysis. The present section describes the raw data collected during the project and the resulting data that serves as a basis for the subsequent discussion.

The raw collected data from its main source, the diary notes, consists of a digital document containing 57,297 words. Their content spans the whole timeframe of the project, which lasted for 11 months. These diary notes are organized in chronological entries spanning 180 workdays of the project. The difference to the 11 months comes from holidays, weekends, and special off-topic days that are not part of the records.

A separate source of raw data used for crosschecking the main source is the detailed table of the working times of the main developer. This table contains entries for 191 workdays. The difference when compared to the diary notes, amounting to 11 days, was crosschecked

between the two sources. It can be attributed to special off-topic days and workdays for which no detailed record of raw data was made. These days especially occurred in the first two weeks of the project when the main developer freshly joined the organization and the active part of the project was not yet fully started.

Furthermore, another source of data was the automated log of the CVS server that maintained the code developed during the project. This raw data consists of entries for 78 days, giving information about the lines of code removed and added each of these days. The number of days covered is less than the half of the raw working days of the project. This was also crosschecked for plausibility. When compared with developer activities extracted from the diary notes, it was confirmed that coding did only start shortly before the first entry of the CVS logs. As code was developed locally and not necessarily checked in on the server every day of the development, the small remaining discrepancy is plausible.

The data emerged from text coding and the pre-analysis of the main raw data source forms the case study "database", which consists of several spreadsheet tables (see also Section 2.3.3). This processed data is organized into three main sets of data: "meeting analysis", "activity analysis", as well as "event chronology and requirements evolution". Each of these data sets consists of entries extracted from raw data following the three different topics. The entries, organized in chronological time-stamped order, refer to the raw information occurrences. For each of these sets, several data fields were defined for which data was extracted from the raw source.

For the subsequent analysis of these data sets, accumulation and aggregation strategies were elaborated by the author. As the data sets are composed of time-stamped lists of entries with qualitative and quantitative information in chronological order, accumulation of the quantitative data over different timeframes and consistent time stamping of qualitative data was applied.

For this purpose, the raw source listing the exact project working times was used. It allowed establishing a consistent timeline for the timestamps of the three data sets by eliminating all non-workdays from the dates. This means that weekends, holidays, and interruptions of the project were excluded from the timeline. The result was a timeline consisting of "day-numbers", denoted as "Day#", which continuously numbers the working days of the whole project. The first day of the project is number one and the last day is number 191, covering the whole timeframe of the project. Based on these day-numbers, also the effective equivalent of workweeks and work-months was calculated. In this way, a "workweek" consists of five working days and they are defined as the "5-Day-number" (i.e. the rounded up day-number divided by five). Similarly, the idealization of a work-month, the "20Day#", consists of four workweeks (i.e., 20 effective workdays). The timescale of effective workweeks (the "5Day#") was found to be the best compromise between the too detailed information on the "Day#" level and the too coarse information resulting from the "20Day#" accumulation.

The timelines were a prerequisite for the data accumulation order to establish time-series, i.e. tracing the evolution of quantitative data fields over time. Notably, the frequency of meetings per effective workweek could thus be calculated and traced over time. If calendar days that were not spent on the project (e.g., holidays, etc.) had not been taken out of the calculation, they would have compromised the interpretation of the data. For example, there would have been a low number of meetings in the time-series during difficult phases of the project as well as in weeks with holidays.

In this way, all data extracted from the raw sources that could be quantified was accumulated for each data field in the three different granularities: per effective workday (the "Day#"), per effective workweek (the "5Day#"), and per effective work-month (the "20Day#"). All data entries, qualitative as well as quantitative, were furthermore time-stamped according to each of these timelines. Measured in these timeline units, the project duration was 191 workdays, 39 workweeks, and 10 effective work-months. This consistent time stamping allowed the interpretation of variations of quantitative data at given phases of the project by referring to events described qualitatively.

The method described here was applied to the three different data sets resulting from the pre-analysis. In addition to these data sets originating from the main data source (the diary notes), the data accumulation over the different timelines was also applied to the CVS server data log. This was at its origin a purely quantitative source and was, therefore, analyzed in a separate data set and it was later joined with the other time-series. All data sets counted together hold 1,137 data entries, each with a time stamp and one or several fields of extracted data. This additional extracted data fields come from the different collected sources. Out of this overall "database", the three most relevant data sets are described in further detail in the following sections.

4.1.1 Meeting Analysis Data

A data set named "Raw Meeting Data" was extracted from the diary notes, which contained special key entries for each meeting that occurred within the timeframe of the project. Furthermore, the entries were crosschecked with hand-taken meeting minutes. The final data set contains 214 entries in form of a table. Each entry corresponds either to a formal or to an informal meeting that took place. Out of these 214 meetings, 201 meetings were identified to be relevant for the project. The others were off-topic meetings concerning other activities, organizational purposes, or did happen without the participation of the main developer and thus without detailed note taking for the study. In addition to these 201 meetings, 54 failed meeting attempts by the main developer have been registered. These failed attempts stand either for informal spontaneous meeting attempts that failed, or for formal scheduled meetings for which at least one party did not show up. For each entry in this data set (i.e., for each meeting), 8 data fields were extracted from the raw data:

1. A unique identifier that points at the raw data entry marked with the GATE tool for text coding (see Section 2.3.3). This identifier relates to the start and end position of the text passage containing the raw data for that entry.
2. The date at which the meeting occurred
3. A list of the participants of the meeting
4. The type of the meeting, i.e., whether the meeting was a formal, scheduled one or a spontaneous informal one
5. The duration of the meeting categorized into one out of four durations: "mini" (up to 10 minutes), "short" (up to 30 minutes), "medium" (up to one hour), and "long" (more than one hour).
6. The topic of the meeting categorized into several categories (including multiple choices): requirements, domain knowledge, design, prototyping, implementation, testing, planning, development process, and finally personal topics. These categories were found to describe best the main topics of the meetings.
7. Meetings occurring as special Scrum meetings were marked as such. Meetings marked with this flag were especially crosschecked with a separate data source consisting of the Scrum "burndown charts"[11] collected in a data set called "Raw Scrum Data". These amounted to 19 entries.
8. A field marking off-topic meetings

Based on this raw meeting data set, data accumulation was performed according to the timelines defined above. Moreover, data aggregation was applied in the form of defining only two lengths of meetings: Mini and short meetings were calculated together on one hand, and middle and long on the other hand. Furthermore, the four meeting durations were weighted in order to provide a single scale of meetings per timeline unit. This scale reflects the approximate time spent on meetings. This allowed generating charts of the evolution of meeting characteristics over time for time-series analysis. The following time-series were established in a new data set called "Meeting Analysis":

- The evolution of the number of meetings per time unit
- The evolution of the average meeting durations per time unit
- The weighted participation of each of the project participants over time
- The evolution of the content discussed during the meetings weighted by duration

These time-series served for the interpretations in the upcoming section, especially for the analysis of the communication issues encountered during this project.

4.1.2 Activity Analysis Data

A second data set was extracted from the main raw data independently of the first one. Instead of focusing on meetings, this data set (called "Raw Activity Data") was extracted using a coding scheme that concentrated on the activities performed by the main developer during the course of the project.

[11] See [76 p. 232] for an example of a Scrum "burndown" chart.

Based on keywords and accounts of activities found in the diary notes, the main activities along with their estimated duration were extracted. For the duration estimates, additional data extracted from the working times table was taken into account. If, for example, an entry of the diary notes explained how the main developer worked on a given day until lunchtime on a spreadsheet for requirements tracking, the time between the start of work on that day and lunchtime would be attributed to the activity category "requirements analysis".

As a side effect, this procedure allowed the estimation of a "data-gap" for the main developer's activities (see also Section 2.3.3). This data-gap indicates which percentage of the working time per day is covered by accounts of specific activities within the diary notes. During data coding, when activities could not be clearly identified, or the duration could not be estimated from the descriptions in their context, no concrete entry of activity could be extracted. This forms a lack of data amounting to a percentage of the total working time of a day, i.e., the data-gap factor. This data-gap can be interpreted as a lack of raw data caused by omissions during data collection or it can be attributed to off-topic activities of the main developer not related to this project and, therefore, not properly reported. The data-gap was used to check the plausibility of the extracted analysis data by combining the data with an independent source (the working times table). For example, a data-gap factor over 100% would have indicated an inconsistency during data collection or data coding.

The raw activity data set consists of 636 chronological entries. For each entry, the following four data fields where extracted from the diary notes:

1. A unique identifier that, exactly as with the meeting data, points at the raw data entry marked within the GATE tool.
2. The exact date at which the extracted activity was pursued by the main developer
3. The type of activity that took place categorized into one out of 19 categories. Additionally, two special categories were reserved for entries extracted together with the activity data treated separately afterwards for aggregation and accumulation. These two types of entries were used for cross-checking meeting data entries.
4. The duration of the activity in form of extractions from time-stamped sequences of events described in the diary notes or through estimations based on the activity descriptions. The durations were cross-checked with the working times table as separate source.

The categories of activities were chosen in order to be as discernible from each other as possible while providing enough fine-grained information. Furthermore, they were chosen to allow aggregation of data into software development phases such as doing requirements, designing, implementing, or debugging and testing. These categories, which were attributed to the data entries using short identifiers for subsequent automated aggregation, can be described as follows:

- Learning or reading of documentation
- Installing or learning CERN-specific development software
- Requirements elicitation and discussion

- Reverse engineering of former implementations of the LRFSC software
- Self-learning about CERN software libraries for control systems
- Requirements analysis based on previously elicited information
- Prototyping for requirements elicitation and analysis
- Prototyping for CERN control libraries
- Working on requirements specification document
- Design of new software
- Design discussion with stakeholders or experts
- Prototyping new client applications
- Implementing new LRFSC client applications
- Implementing new server side FESA class
- Debugging and testing of new LRFSC software
- Debugging CERN specific libraries
- Documentation for new LRFSC software
- Planning Scrum sprints or preparing Scrum meetings
- Other activities not related to the project

In addition, the two special categories for separate entries were for failed meeting attempts and occurrences of meetings described as especially important by the diary notes.

This raw extracted data set was processed into the "Activity Analysis" data set through accumulation and aggregation. The accumulation over the different timeline units was executed analogously as for the meeting analysis described above. Additionally, aggregation was performed for activities belonging to a certain phase of a software development process. These phases were *doing requirements*, *design*, *implementation*, and *testing*. Separately, the following types of activities were aggregated: *prototyping*, *learning*, or *other activities* as well as *Scrum-related activities*.

Out of this accumulated and aggregated data set, the following time-series were constructed:

- The evolution of the time spent on activities related to the different phases of a classic development process.
- The evolution of time spent on the separate types of activities.
- The evolution of a "focus-factor", which was calculated by comparing the aggregated durations per time unit of activities directly related to the software development to the overall working time per time unit.

These time-series were considered for further interpretation, especially for gaining insights about development process issues encountered during this project.

4.1.3 Event Chronology and Requirements Evolution Data

The third important data set extracted from the data sources consists mainly of qualitative data joined with a few fields of additional quantitative data. For this data set, a chronological sequence of two different types of events was manually extracted from the diary notes: key events that had an impact on the project's evolution or outcome and new and changed key

requirements. Both were extracted by marking relevant text utterances in the diary notes using the GATE tool. Then a short qualitative summary of each requirement and key event was extracted. In total, 325 entries were extracted into this raw data set, from which 117 were requirements entries and 208 were event entries. Each entry of this "Raw History Data" contains the following six extracted data fields:

1. The type of the entry: event or requirement
2. The exact date of the occurrence of an event or the date at which a requirement was first elicited or changed
3. Flags denoting important or special events
4. The source of a requirement or the persons related to an event
5. A short description of the event or requirement
6. A development process phase attributed to events based on their implications (see description of the previous data set)

For all entries concerning requirements, the following additional five quantitative data fields were extracted (by combining different independent sources):

1. A tag assigning one or more categories to a requirement: These categories denote which part of the LRFSC system was concerned by the requirement.
2. A cross-reference identifier pointing to the requirements entries in this data set related to the given requirement. This allowed retracing of the evolution of certain requirements.
3. A flag signaling whether a requirement entry stands for a newly elicited requirement or for the change of a previous requirement
4. A flag indicating if the given requirement was satisfied fully or partially by the final software solution elaborated during this project. For this data field, the final software was analyzed which thus served as additional data source.
5. A reference in the case that the concrete requirement has been part of the requirements specification document elaborated during the project.

This raw data set served for the subsequent analysis in two separate ways:

First, all events described by the diary notes that turned out to be important for the course of the project were marked as such in the data set. Through the consistent timestamps, these marked events could be used for the interpretation of findings originating from the other data sets. For example, one key event was that the section leader (SL) urged the main developer (MD) at one point to start "coding". This lead to an observable change in the activities data set. Moreover, such key events were often corresponding to or related to decisions taken during meetings. Therefore, these qualitative entries describing the events allowed a cross-check with the quantitative extractions of the meetings data set.

Second, the quantitative data of this data set was also processed using accumulation and aggregation as described earlier on. For event entries, the project phase or the activity attributed to the event was accumulated over the different timeline units. This helped to

cross-check the separately extracted activity data in order to identify possible inconsistencies. For requirements entries, the following time-series were constructed out of the accumulated and aggregated data:

- The evolution of the project participants as sources of requirements
- The evolution of the number of requirements according to the different attributes: *new*, *changed*, *successful*, and the total *number of requirements* per timeline unit

The additional information provided by this data set and the resulting time joins the other data sets in providing the basis for further analysis of the project under study.

4.2 Communication Issues

The communication between the participants of the project under study was characterized by several specific issues that strongly affected the project's progression. Especially the main developer was directly concerned by the issues since he had to collect all necessary information that would allow him to gather and understand the software requirements and to develop a new software solution. These issues became apparent during the course of the project; however, they were poorly understood at that time. Only through the retrospective analysis, the influence factors surrounding the communication issues could be properly identified. This analysis is based on the quantitative and qualitative data presented above.

In the project, the main developer (MD) had the task to identify the requirements for the new software solution, to design it, and to implement it as well. Since he came to CERN without having prior knowledge about software development within CERN's technical context (described in 3.1.2), he needed to gather all necessary information during the timeframe of the project. This defined the principal mainspring for the communication between the project participants. Consequently, at many occasions, the MD was the driving force behind meetings that took place to advance the project. Hence, he was the first to experience issues with the communication among the project participants. It should be noted that the term “meeting” used in this study does not only refer to formal, scheduled meetings but to every relevant interaction that occurred between the MD and other project participants.

A first interesting finding of this retrospective analysis is that the predominant number of interactions that happened between the MD and other project participants was informal in nature and of very short duration. In total, only 39 meetings (19%) out of the 201 relevant meetings were formal. From all relevant meetings, 85 lasted only up to 10 minutes (42%) and in total 144 out of 201 meetings lasted shorter than 30 minutes (72% of the relevant meetings). It is also interesting to relate both aspects: Only 19 out of 57 meetings (i.e., 33%) that lasted over 30 minutes and only 7 out of 16 meetings (i.e., 44%) lasting over one hour, were formally scheduled meetings. This indicates that informal meetings and spontaneous interactions played an essential role for the project. In total, over twice as much communication between the MD and other participants happened through informal interactions than through formal and scheduled meetings.

The predominance of meetings of short duration was not equally present during the whole project. Figure 16 shows a plot of the time-series of the overall number of meetings, grouped in two different durations, over time. It shows that only during the first effective workweeks, the majority of the meetings were meetings lasting longer than 30 minutes. With the exception of workweek 14, the highest meeting frequency happened between effective workweek number 28 and 38, when short meetings were largely predominant.

Based on qualitative descriptions from the raw diary notes, it can be interpreted that the first weeks were characterized by an "acclimatization" of the main developer who freshly joined CERN for this project. The few first meetings were scheduled for the MD to get him in touch with the environment and allowing him to gather first information about requirements. Only as soon as the constellation became "settled", spontaneous meetings started to take place.

The high number of meetings, and especially of short meetings, during the last third of the project's time can be explained by the qualitative data from the chronology of key events: In workweek number 26, the Section Leader (SL) suggested trying the agile Scrum method for the remaining time of the project. It was started in workweek number 26 and effectively introduced in workweek 27. It finished shortly before the project's end in workweek number 36. Important parts of Scrum are frequent and regular short meetings called "Daily Scrum" and the fostering of informal interaction among the developer team. This was the major influence factor for the evolution of the meeting frequency.

Figure 17 plots the time-series showing the averaged duration of meetings over time. What is clearly apparent in this plot are the two periods of extremely low communication. During workweeks 9 and 22 up to 24, no communication happened at all. Data showed that in the

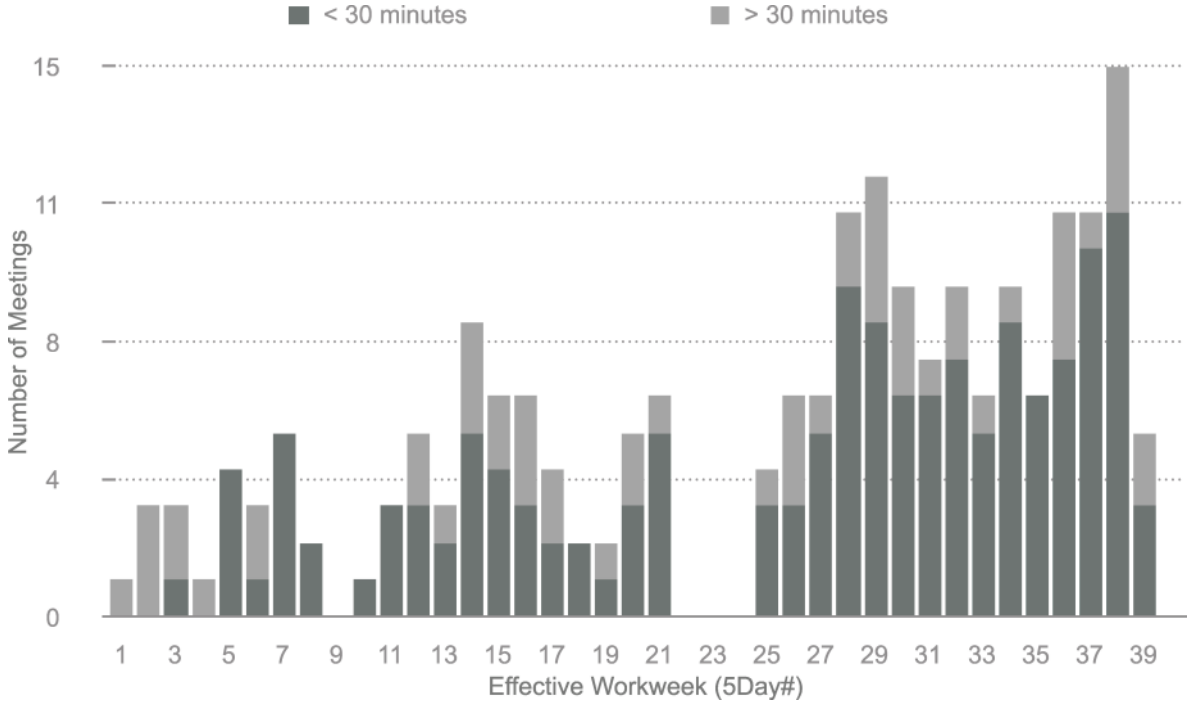

Figure 16. Time-Series Chart of the Number of Meetings per Effective Workweek

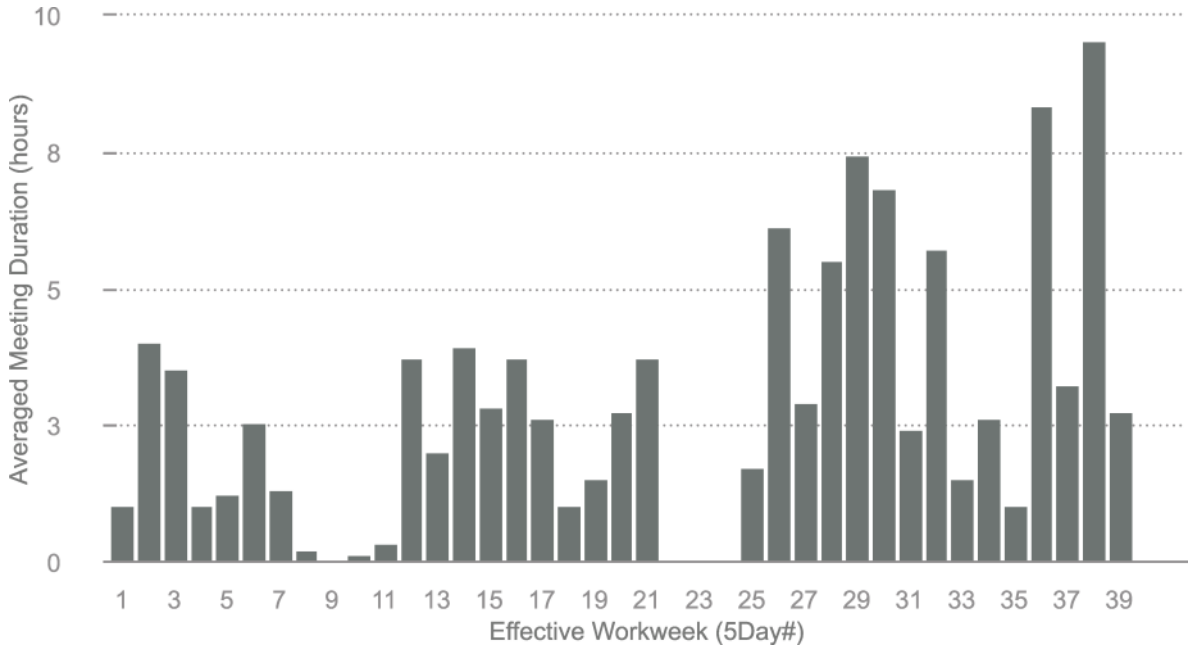

Figure 17. Time-Series Chart of the Averaged Meeting Duration per Effective Workweek

weeks 9 and 10, no single interaction or communication happened during eight consecutive days. Furthermore, between weeks 22 to 25, no project-related communication took place during 16 consecutive effective workdays. This can be interpreted as two complete communication breakdowns since even very short informal interactions of fewer than ten minutes in duration would have been registered. Especially the second breakdown lasting for 16 workdays is remarkable, as all project participants besides the Machine Expert (ME) were located in offices on the same floor close to each other. This should have naturally led to frequent informal interactions [44 p. 136] as it did during other phases of the project. Taking out the two periods of communication breakdown, an average of six meetings occurred per workweek corresponding to an estimated average meeting time of 3.3 hours per workweek. Comparing the three workweeks of no communication justifies the relevance of this breakdown.

The first minor communication breakdown can be explained by the key events that happened in the frame of the elaboration of the requirements specification document by the MD. The MD started the work on a requirements specification document for one of the new GUI client applications to be built at the beginning of workweek 8. At first, the MD elaborated a glossary document specifying all specific terms of the project's technical context such as "cycle" or "PPM" (see 3.1.2). In order to check whether his knowledge and his assumption about those terms were correct, he requested feedback from the other project participants. This had been necessary for the document beforehand since most requirements elicited until that time made explicit reference to those terms. Unfortunately, the MD had to wait four workweeks to get feedback about this glossary document (which happened finally in workweek 12). During this waiting time, the MD had to continue his work on the requirements specification document despite missing the essential feedback about his understanding of the domain.

Retrospectively, this rather formal act of submission of a document for explicit review had caused the blocking of the communication in the first place. The participants having received the document for review did not actively initiate interaction with the MD as long as they had not looked through the document yet. On the other side, the MD had no incentive to contact the other participants since he already had requested feedback from them and was waiting for reply. This constellation seemed to have hindered any informal contacts. Since no formal meetings were scheduled during this time, no communication took place at all.

Besides this first short communication breakdown, the second was more relevant to the project's evolution and lasted for a significant time. As described and depicted above (see Figure 17), during 16 consecutive workdays, no project-relevant interaction occurred between the MD and the project participants. A possible explanation was again found in the qualitative chronology of events.

The implementation phase of the project began in workweek 17. It was caused by a key event, according to which the MD was urged to begin with the implementation since the other participants judged that all requirements should be known by that time (this turned out to be not correct as requirements emerged or changed even towards the end of the project). The first approach of the MD was to transform low-level prototypes into an interactive prototype in the Java programming language. The work on these prototypes proceeded until the MD ran into a lack of necessary knowledge about CERN's client-side middleware and the overall system design. Since no other participant was available to provide the missing knowledge and no detailed documentation existed, the MD tried to reverse engineer the former software solution as well as parts of the middleware to understand how to employ it properly. Then, between workweek 17 and 22, 16 requirements emerged during meetings with the SL and the ME. This worsened the situation of the MD, who already struggled with the limited knowledge about the specific systems in order to come up with a design that would satisfy the previous requirements. The fact that no proper “design” phase was foreseen, as the MD had initially planned, added to the difficult constellation.

This finally led to the situation that the MD got blocked in his further progression by the lack of specific domain knowledge and the lack of a source for this missing knowledge. Since the MD tried to resolve the situation first through reverse engineering, no communication intents happened from his side. Finally, in workweek 25, the MD had the chance to inform the SL about the issues, as he could find no solution viable in the longer term. During this decisive meeting, the MD was informed that this project would get more attention from that time on by the other participants, since they were occupied with other higher priority projects before. This aspect of different “priorities” explained the long communication breakdown from the other participants’ side. This important meeting of workweek 25 then led subsequently to several important changes in the course of the project.

Therefore, because of the second communication breakdown, the following changes occurred to the project. The changes were:

- Introduction of the Scrum process, which was tried for the first time by the project participants on this project. As a consequence, this forced fixed meetings on a regular basis to foster interaction and avoid further breakdowns.
- Addition of two experts (the LE and the HE) to cover the missing domain knowledge for design and implementation
- Extension of the project for one month to give the project more time under this new setting (initially only 10 calendar months instead of 11 were planned).
- The MD abandoned his formal document-driven approach of doing requirements in favor of a more informal approach to break the lockdown in the project's progression and to better fit the "way of working" in the given environment.

The consequence of adding the two experts to the project to profit from their specific knowledge was also measured by the quantitative meeting analysis. Figure 18 shows the time series established based on the quantitative data from the meeting analysis data set combined with qualitative data from the event history. The graph represents on five horizontal time axes (in effective workweek numbers) the weighted participation of the five different project participants in meetings with the MD. Since the exact amount of communication is not as relevant as the proportion of the different participants and the evolution of the intensity, the representation using the 5-axis bubble chart was found to provide the best basis for interpretation.

Because of including the two experts, the HE and the LE, a considerable amount of communication now occurred with these experts. This can be seen in Figure 18 as from workweek 26 on, when they effectively started their participation. The communication with these experts played an important role until the end of the project as shown in the chart. This

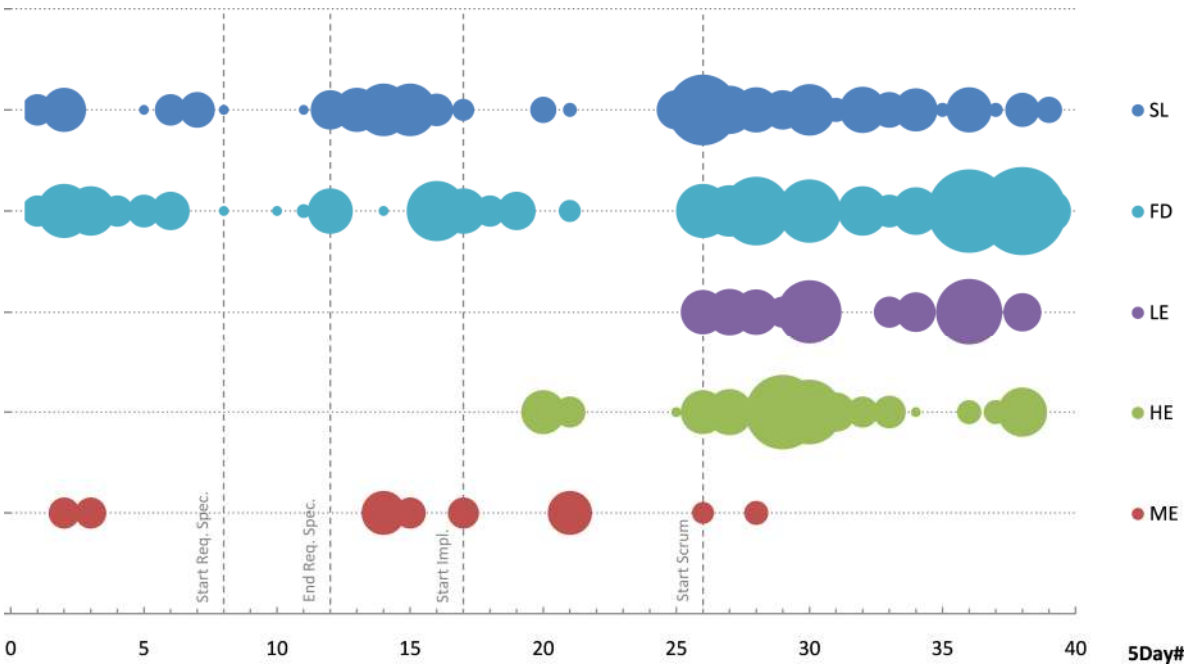

Figure 18. Time-Series Chart of the Meeting Participation per Effective Workweek

reveals the importance of the decision of including them in the project and shows that exchange between the MD and the two experts played an important role.

The time-series plotted in Figure 18 reveals also an additional finding. The axis corresponding to the ME on the bottom of the figure shows a dispersed and relatively low amount of communication compared to the other participants. This is interesting as the ME was the only participant representing the "clients" (see 3.1.1.2).

According to the meeting data set, 7 out of 9 meetings that took place with the ME were formal meetings. The predominance of formal meetings can be explained by the fact that the ME was the only participant of the project who was not part of the organizational unit in which the project took place. Moreover, his office was located on a different site of CERN and, therefore, spontaneous meetings rarely occurred (see [44 p. 136] for the relation between proximity on interaction frequency). Furthermore, the low rate of interaction with the ME had other reasons according to the collected qualitative data. The ME had apparently had personal issues with the SL, that originated from a time predating the project. This might justify the rare occurrence of meetings with the ME, as the SL had coordinated the meetings with the ME in the frame of the project[12]. This issue emerged at several occasions which were revealed by the following key events:

- In workweek 17, the SL decided that the ME would not be asked anymore about requirements or feedback. In his view, the involvement of the ME had only slowed down the progression of the project.
- During workweek 20, the MD challenged the SL's previous decision and asked the SL if the ME should not be involved in a review of the visual software prototype before too much time would be invested in its implementation. The SL declined, as his experience, especially with the ME, showed that it was better to implement things first by focusing on getting a technically simple and feasible solution and then confront the client with the implementation as a fait accompli.
- In workweek 21, the ME was asked to provide a short specification or at least a technical description concerning the new resonance control mechanism that had to be added (it had to interface with special equipment known to the ME). Unfortunately, the ME was very reluctant to do so. Even after several requests from the SL and the MD, the ME had not provided the necessary information. Only in workweek 26 (five effective workweeks later), he agreed to a meeting held especially for this purpose.
- In workweek 25, the SL complained to the MD about the attitude of the ME who showed, in his view, too little commitment to this project even though the project was concentrated on realizing the ME's wishes.

Finally, another communication issue concerning the Former Developer (FD) was identified based on qualitative and quantitative data. The FD showed at several occasions a reluctant attitude towards formal meetings. He explained this by his bad experience with several

[12] As pointed out by [2]: "[…] organisational and social issues have great influence on the effectiveness of communication."

regular meetings in which he had to participate that gave him personally the feeling of "losing time"[13]. This reluctance towards formal meetings was also reflected in quantitative numbers. In total, 76 out of the 105 meetings that took place with the FD's participation were informal meetings. On average, this makes 45 hours of informal interaction out of the total of 64.5 hours, which corresponds to 70% of the FD's communication time on the project. In comparison, only 48% of the communication time of the SL came from informal meetings. This resulted in different communication "habits" the MD had to follow, depending on the participant.

4.3 Domain Knowledge and Requirements Issues

The communication issues reported above were related to issues concerning domain knowledge and requirements. Based on the qualitative and quantitative data presented in 4.1, these domain knowledge and requirements related issues are addressed in this section.

Figure 19 is a plot of the time-series showing the evolution of the number of new and changed requirements as they were elicited during the project. What is remarkable in this data is that requirements seem to have emerged during the whole timeframe of the project. Moreover, key requirements that were already elicited kept changing almost until the end of the project. Even in workweek 36, close to the end of the project, such changes occurred. Figure 19 furthermore reflects once more the two communication breakdowns, as the elicitation of requirements is bound to direct interaction if stakeholders are the primary source such as in

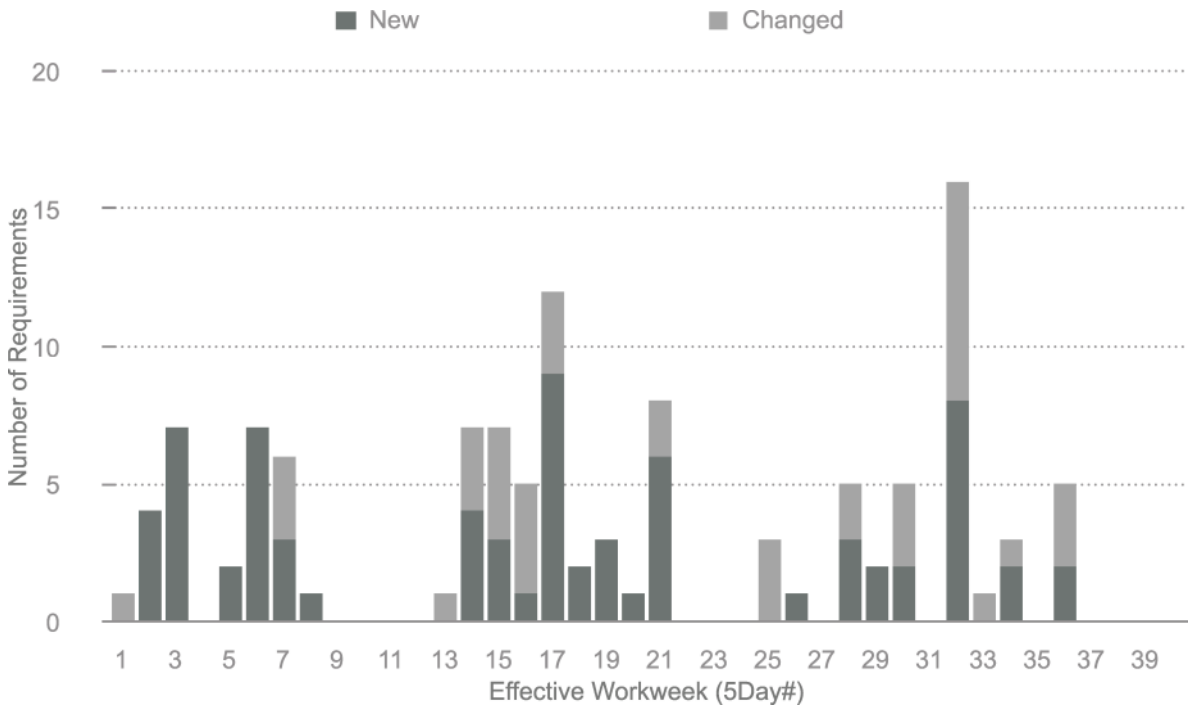

Figure 19. Time-Series Chart of the Number of New and Changed Requirements per Effective Workweek

[13] This effect has been observed previously and reported in the literature. See, e.g. [44 pp. 215-16].

the present case.

No written requirements specification existed for the project at the time it had started. The MD had the task to undertake the requirements elicitation for the new software package. The initially developed software solution was at least partially available in form of source code and could thus be used to reconstruct some former requirements. Furthermore, for this purpose, the FD and the SL were available as the former developers of the initial software.

The first approach of the MD for requirements elicitation was to interview the FD, SL, and the ME. For this purpose, the MD used a custom spreadsheet for the management and tracking of the elicited requirements. Based on this information, the MD elaborated a requirements specification document that he submitted to the FD, SL, and ME for review and comments at the beginning of workweek 12. Unfortunately, at the end of workweek 13, it became clear that no review of the specification would take place. The SL had looked through the specification, but the FD and the ME showed a reluctant attitude towards the document. The MD could convince the SL and the FD to give concrete feedback about the glossary of specific terms that were an important part of the specification document, as it showed whether or not the MD had correctly interpreted the domain-specific concepts around which the requirements evolved (e.g., cycle, pulse, PPM, etc.). The request for the glossary review had come from the MD already at the beginning of his work on the specification document in workweek 8, but the feedback was only provided after the document was finished. The feedback finally revealed that many essential terms were misunderstood by the MD and that invalidated essential parts of the specification. Besides the glossary, the document was never formally reviewed.

As a consequence, the MD eventually abandoned the initially intended "way of working" with specifications. Already at the end of workweek 13, he started to focus on the development of visual mockups (low-fidelity prototypes) of the client applications that had to be redeveloped. This allowed immediate and spontaneous feedback from the other project participants. The concept was well received as the MD initiated short walkthrough meetings of the mockups (they were printed out on paper for drawing on them) with the SL, the FD, and the ME. Feedback and newly elicited requirements based on these walkthroughs were integrated into the prototypes on an iterative basis. In this way, especially some of the not well-understood requirements from the initial elicitation could be clarified. This procedure led to several changed and some newly added requirements.

This new "way of working" on the requirements elicitation then changed in workweek 17 when the MD was urged by the SL to start the implementation. The SL judged that all requirements should have been settled at that time. Apparently, the iterative process of eliciting the requirements through realistically looking low-fidelity prototypes mediated that all requirements and even the software design are fixed and that the subsequent implementation will be an easy task[14]. This impression was also highlighted during a meeting in which the

[14] This effect of prototyping has been observed previously: "[…] because visible outputs are quickly available, users and managers are easily seduced into believing that the design/code/test phase as well as the modeling phase can be skimped on." [35]

SL judged the project to be "ahead of time" based on the given prototype evolution. However, as depicted in Figure 19, changes in requirements occurred almost until the end of the project. The requirements were thus not fixed by the prototype when implementation started. These requirements mainly concerned the application logic operating behind the user interface.

An important issue was that the requirements elicited during the project were very "technical" by their nature, i.e., strongly oriented towards a solution instead of describing the problem. In effect, this issue was also reflected in the fact that requirements emerged and changed at any phase of the project.

An explanation of this issue can be based on the essential context of this project, which was the predominance of diverse domain specific knowledge that was necessary but initially missing for the MD (see Chapter 3). The amount of specific organization- and system-dependent knowledge was especially high and could only be acquired from the other project participants[15]. Domain knowledge played such a major role that even the most basic requirements of the redevelopment project could only be properly understood by possessing some of this domain knowledge. Not only "business domain knowledge" relating to what the "procedures" and "products" of CERN are, was necessary (i.e., the operation principles of the accelerator complex and the beam production), but also "technical domain knowledge" about the existing system and its related technical infrastructure was needed for the proper understanding of the requirements (see Section 2.2.3.2 concerning the domain knowledge differentiation).

Unfortunately, the MD was initially not explicitly aware of this issue, which led to the situation that requirements elicited and formalized by the MD were inherently confounded with domain knowledge. The knowledge that the MD acquired during the course of the project about the requirements was directly bound to the acquisition of domain knowledge. The high number of occurred requirement changes can hence be explained by the situation that the MD's acquisition of domain knowledge dragged behind the acquisition of knowledge about the requirements. This did not allow the MD to properly understand and specify the requirements. In the given constellation, the MD could acquire necessary domain knowledge only reactively instead of proactively. Only the lack of given parts of the domain knowledge revealed what knowledge was concretely needed. This eventually led to the addition of the two additional experts (the LE and the HE), as essential domain knowledge was found to be missing[16].

As mentioned in the previous section, the MD had initially a different approach towards doing requirements and communicating about them than the other project participants. Besides not "speaking the same language" by missing the relevant domain knowledge, the MD pursued a formal document-driven approach. This differed from what the other participants were used to. The MD's initial effort was directed towards acquiring and retaining knowledge about high-level needs describing the problem domain of the software to be built. However, the SL,

[15] The knowledge existed mainly in form of organizational "folklore". See [106] and 2.2.3.2.
[16] The benefit of adding domain knowledge experts was also suggested by [109]. See 2.2.3.2.

the ME, but especially the FD, were used to focus directly on the implementation for developing a software solution for a given problem. A consequence was that questions from the MD targeted towards "needs" in the problem domain were answered in terms of the solution domain.

These differing approaches can be illustrated by one event that happened in workweek 6: The MD went to ask the FD about a requirement coming from the ME concerning a certain "constant amplitude and phase mode" that the MD did not understand. The approach of the FD was to assume that the requirement is "important" and then think through how the new software could be designed and implemented in order to fulfill it. This would then explain the requirement and show if it would be feasible. If not, it should be rejected.

The effect of those differing approaches was that many requirements were strongly tied to design and implementation decisions. Hence, the problem and the solution domain became confounded. This can be seen on the example of the essential key requirement about the "device working in PPM mode" (see 3.1.2) that emerged already in workweek 6 (already initially confounded with a design solution) and reemerged in many meetings and kept changing and influencing other requirements until workweek 36 (i.e., during 30 effective workweeks). The retrospective analysis of the requirements evolution showed that many statements of needs that were called "requirements" during the project by the other participants mostly related to the system design and thus to the solution domain.

This fundamental issue concerning the project's requirements and the lack of domain knowledge on the side of the MD led to a lock-down in the project in conjunction with the second communication breakdown. It was finally overcome through the adoption of development approaches by the MD that suited the given context and environment and by the addition of two additional experts holding essential domain knowledge. This closely relates to the development process issues that are discussed in closer detail in the following section.

4.4 Development Process Issues

The analysis and interpretations of communication issues as well as issues relating to requirements and domain knowledge above highlight the presence of issues concerning the development process. Extending on the discussions of the previous sections, this section addresses those process-related issues based once more on the qualitative and quantitative data presented in Section 4.1.

Two different time-series were established that show the evolution of "traditional"[17] development process activities, which were idealized into four essential categories for this purpose: doing requirements (REQ), software design (DES), implementation (IMPL), and testing (TEST). Both time-series are shown as multiple-axis charts in Figure 20. The upper chart shows the evolution of the MD's activities categorized into these phases, whereas the lower chart shows the main discussion topics of the meetings also categorized into these

[17] Also called "tayloristic", see [32]

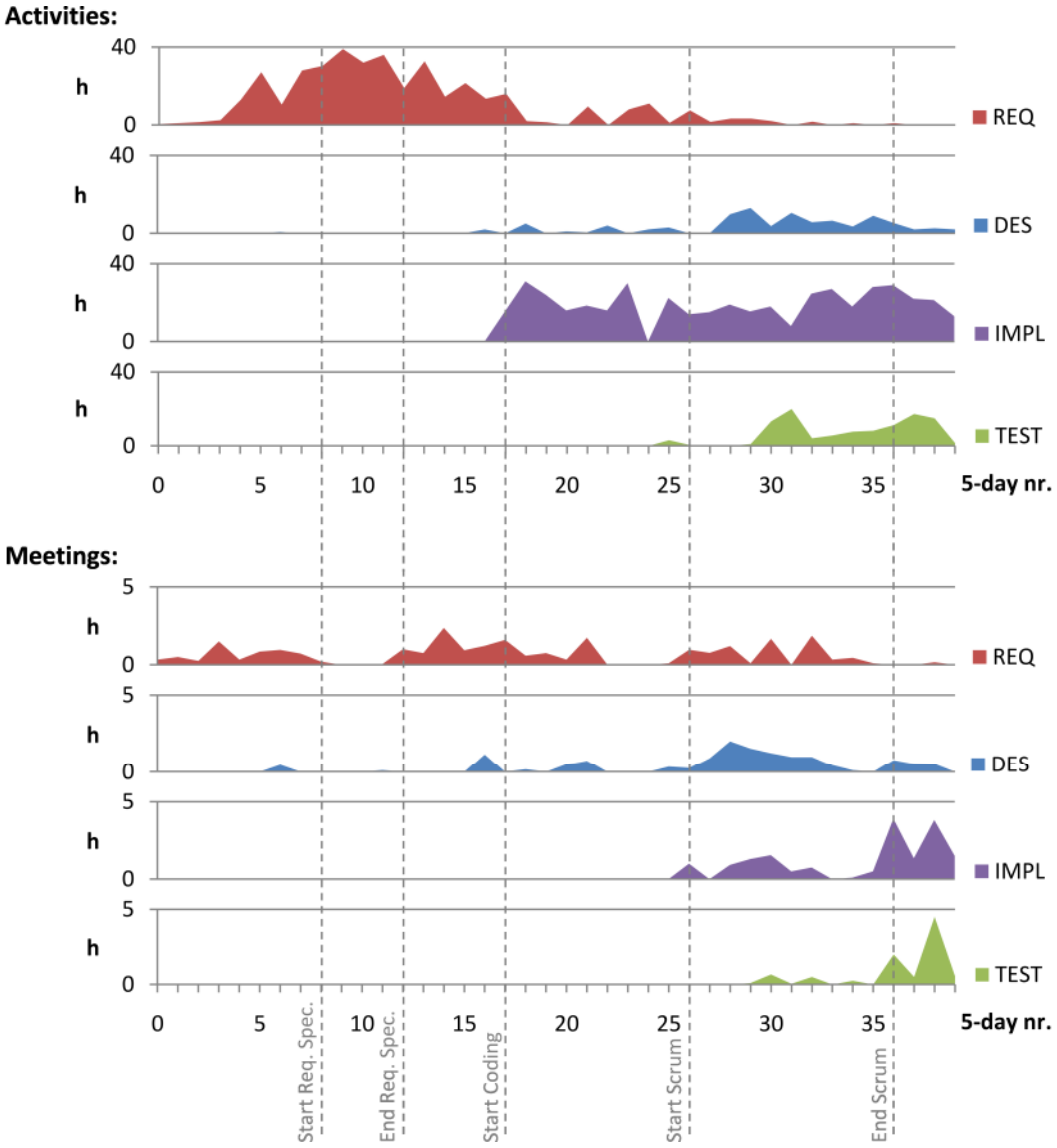

Figure 20. Time-Series Chart of the Activity and Meeting Phases per Workweek

phases. Both charts are aligned on their horizontal time axis to combine both time-series. They both use the same dimension on the vertical scales indicating the estimated time spent on the given phase within each effective workweek. Furthermore, vertical dotted lines are overlaid that denote relevant key events from the event chronology for further interpretation.

The evolution of the phases depicted in Figure 20 confirms that "doing requirements" was dispersed over the whole timeframe of the project. However, a finding is that even though requirements were discussed almost until the end of the project during meetings, they preoccupied the MD (the only project participant working full time on the project during the whole timeframe) intensely only until workweek 17 and then occasionally between workweeks 21 and 29. This difference can be explained by the change of dealing with requirements that occurred as the MD abandoned the initial document-driven approach. Initially, the requirements were elicited by the MD through discussions and interviews with the other participants. They were then formulated, analyzed, and tracked, and finally specified

into a requirements specification document. This procedure was time-consuming as shown by the intense REQ phase in the upper chart of Figure 20. As explained above, the complexity of this undertaking was caused by the different "languages" of the MD and the other participants and the MD's necessity of "reactive" domain knowledge acquisition.

As soon as the attempt to get the requirements specification document reviewed failed, the MD started to switch to working with iterative refinements of visual low-fidelity prototypes (mockups) of the client applications between workweeks 12 and 13. This approach necessitated also considerable amount of time to analyze new requirements and requirements changes gathered during walkthroughs of the prototypes with the other participants. The changes had to be integrated into new iterations of the prototypes. This is reflected by the still considerable amount of time spent on doing requirements between workweeks 13 and 17 and correlates with an increased amount of meeting time concerning requirements between workweeks 14 and 17.

As explained, the SL had urged the MD to start the new software implementation in workweek 17. As the MD followed this request, it led to a considerable amount of work time spent on implementation efforts (IMPL) instead of the time spent on doing requirements (REQ), as can be seen in the upper chart of Figure 20. What can be called the traditional "requirements phase" of the project was "shut down" nearly at the half time of the project[18]. This sudden start of implementation led to an omission of any preliminary design phase (see the low design-related activities and meeting topics between workweeks 17 and 25) that would have followed the requirements specification before starting implementation.

Retrospectively, it turned out that the different "approaches" towards software development of the MD and the other participants were an issue causing the project progression downturn leading to the second communication breakdown. The MD had started the project with the intention to proceed according to a formal document-driven approach and a sequential development process. Contrary to this, the other participants were used to following an informal "code-and-fix" approach to software development. This difference in the approaches turned down the progression of the project at a decisive moment. Together with the additional issues described above (Sections 4.2 and 4.3), the communication broke down.

The initially missing time devoted to a "design phase" after doing requirements can be found represented in both charts starting with workweek 27. At this time, the agile Scrum methodology was tried out on this project. It finally involved actively all project participants and served as incentive to conduct regular meetings. Scrum is based on a series of consecutive "Sprints" during which a subset of "user stories" (a form of requirements representation) is selected to drive the design and implementation effort[19]. This resulted in a mixture of all four different categories of activities and communication topics during the Sprints (see Figure 20). The initially high individual time spent by the MD on "doing requirements" was transposed to

[18] This phenomenon of shutting down the "requirements phase" at half time was already observed in other studies. See [109 p. 68].
[19] See, e.g., [54], [90], or [33] for information about Agile Methods and Scrum.

time spent on this activity during meetings. Knowledge about requirements in form of "user stories" was extensively discussed in the frame of the Scrum meetings.

The reflections about the changes in the development approaches during this project can be summarized by splitting the project into three parts. These parts can be time stamped within the project chronology and described as follows:

1) The first part was characterized by a document driven approach. It consisted of requirements elicitation through interviews and discussion followed by the elaboration of a requirements specification document submitted for review. On this last point, this approach failed in workweek 13.
2) The second part was then the iterative approach of using low-fidelity prototypes of the client applications to gather feedback and to elicit new requirements through interactive walkthrough meetings. This approach could not be pursued as soon as the application logic of the new software package had to be designed and implemented since essential domain knowledge was missing.
3) As the MD was missing essential domain knowledge that could not be gathered through the review of prototypes, the third part of the project started in workweek 25. It was characterized by an agile development approach based on Scrum, in which all the active participants of the project were involved. Before this final part, the development effort was a mainly individual effort of the MD. Only in this last part, this changed into a collective effort of all project participants.

Together with the closely related issues concerning communication, requirements, and domain knowledge described in the previous section, this discussion provides an overall analysis of the specific issues that occurred in the frame of the project under study.

5 LESSONS LEARNED

This chapter distills the lessons learned for the retrospective analysis of the project under study. Although the project took place in a very specific setting, some of its characteristics are comparable to projects in industry (see 2.2.1). Based on the specific characteristics described in Chapter 3 and the data about the project analyzed in Chapter 4, the findings of this retrospective case study are generalized in this chapter. Following the initial research topics (see 2.2.2), lessons learned concerning the development process, the requirements engineering, and the communication within the project are discussed.

First, the impact of domain knowledge issues on requirements and the development process are addressed. This is followed by discussions about evolutionary prototypes as requirements representations and, finally, about the fostering of knowledge exchange.

5.1 Domain Knowledge and the Development Process

The analysis of the case under study revealed the importance of issues with domain knowledge for the development process and especially for doing requirements. The relevance of domain knowledge for successful software development and especially for doing requirements is known in the literature (see 2.2.3.2). In short, developers need to hold all necessary technical and application domain knowledge to successfully conceive and develop software. When this prerequisite is not true, as in the given case, different issues can arise.

The most relevant source on this topic is the study by Curtis, Krasner, and Iscoe [42] that revealed three essential issues in software engineering projects (see 2.2.3.2). These issues were also observed in the present project (see Chapter 4): the "thin spread of application domain knowledge", "fluctuating" requirements, as well as communication breakdowns. Curtis et al. [42] note that the domain knowledge issues are especially characteristic of projects dealing with software developed for a system embedded in a larger infrastructure, which is exactly the case in this project. Moreover, they found that the missing domain knowledge of developers was a reason for requirements fluctuations because of "[…] an

incomplete analysis of the requirements" [42 p. 1275], which was also observed in the project at hand. In this way, major findings from [42] were essentially confirmed by the present case study. The literature overview in 2.2.3.2 further suggests that requirements engineering can be hindered when issues with domain knowledge are not explicitly tackled. This was confirmed by the present study.

In the present case, as described in Chapter 3, a large amount of diverse domain knowledge was involved. The knowledge was dispersed unevenly among the different participants of the project. No appropriate documentation existed that could provide the necessary knowledge. These circumstances influenced the development process of the project under study.

The first part of the development project was characterized by a document-driven approach based on direct and explicit requirements elicitation from the "stakeholders". At that phase of the project, those stakeholders were the ME in his role of an expert user and representative of the operators, the FD as a source of requirements of the previous implementation, and the SL as the person responsible for the project. As unveiled in Chapter 4, this initial approach of getting a set of agreed requirements from those stakeholders through a specification document review had failed.

In the second part of the project, the initially pursued approach of "eliciting requirements from stakeholders in order to produce a written specification" was abandoned. It was found that focusing on "who holds which necessary knowledge" was far more relevant than documenting "who had which stake". In this way, a key to the final success of the project was the identification of missing sources of domain knowledge, which led to the subsequent addition of two domain experts (the HE and the LE).

A lesson learned can be generalized as follows: If domain knowledge plays a predominant role (e.g., in projects that relate to complex existing technical infrastructures) and it is "thinly spread" among the developers (following [42]), focusing on *knowledge holders* can be more important than focusing on the roles of *stakeholders*. Identifying who holds which essential knowledge about the domain or the requirements and transferring this knowledge to the developer(s) is the key for the understanding of the needs and for the elaboration of a software solution satisfying those needs.

The approach initially pursued by the MD corresponded to a document-driven approach of a sequential development process. The first step would have been the elicitation and specification of requirements, which would then be reviewed. Based on the agreed requirements, a design should have been elaborated that would have served to define the implementation of the new software solution. However, this approach failed in the "requirements phase" and on the switch towards the "implementation phase". The project was close to become a failure. It turned out that pursuing sequential development phases was not applicable in the given context.

When developers are missing essential domain knowledge and efficient knowledge transfer does not take place, a sequential succession of doing requirements, the design, and the

implementation becomes unfeasible. This lesson learned is not new and was found in a recommendation from Boehm [16]:

> "Avoid using a rigorous sequential process." [16 p. 24]

This is complemented well by a finding of [42 p. 1284] that was already presented in Chapter 2:

> "[…] developers held a model of how software development should occur, and they were frustrated that the conditions surrounding their project would not let them work from the model."

In the present case, the MD had to abandon his idealized conception of how he thought software could be developed in this project. Hence, this was one of the major "personal" lessons learned by the MD in the frame of this project. This lesson might turn out to be helpful in other software development projects as well.

5.2 Evolutionary Prototypes as Requirements Representation

The analysis of the project under study discusses the failure of the initial approach of specifying the software requirements in a written requirements specification document. The case showed that the initially pursued document-driven approach failed as no review of the elaborated requirements specification document could be achieved. Instead, an iterative incremental approach using evolutionary low-fidelity prototypes proved to be more successful for eliciting and reviewing the requirements in the given case.

In the project, an iterative incremental approach with visual prototypes was adopted by the MD as the document-driven approach failed. In this approach, the knowledge about the domain and the requirements acquired by the MD was captured within low-fidelity prototypes of the GUI applications. The prototypes became artifacts making the knowledge about the needs and the domain explicit. Through walkthrough meetings of those prototypes, the MD could identify mistakes or omissions in his vision about the problem and the solution to be built.

The case shows, that in an organizational context in which document-driven development is not common, another form of representation of requirements should be considered. Using low-fidelity prototypes as requirements representation led to a better acceptance under the given conditions and necessary feedback about the visualized requirements was gathered. Nevertheless, the employed approach also had its flaws. In this case, the prototypes could only represent requirements concerning some parts of the system. Requirements concerning only the underlying application logic could not be visualized and remained difficult to be addressed and reviewed.

In this reasoning, there is an inherent differentiation between requirements and their representation [59]: When doing requirements, not the requirements themselves are

manipulated by the developers when specifying them in a requirements specification, but only representations of the requirements are. Not every form of representation works in every situation and therefore, the right form of representation should be carefully chosen.

In the case under study, it was found that representing the requirements in the form of prototypes was more successful than representing them in a written textual form. In the development culture given by the organizational context of the project, the written document led to reluctance from other project participants who were not used to document-driven approaches. Under such circumstances, it may be useful to abandon a predefined form of requirements representation if it proves to be not appropriate. This can help to prevent a looming failure such as in the present case. For some forms of representation such as low-fidelity prototypes, this might include not being able to explicitly differentiate between knowledge about requirements and domain knowledge.

5.3 Fostering Knowledge Exchange

As described above, the main software developer had initially no context-specific technical or business domain knowledge at all, as he joined CERN exclusively for this project. Moreover, this lack of necessary domain knowledge hinders productive development activities. Together with the identification of the key knowledge holders, the transfer of domain knowledge is a prerequisite and has to take place.

The analysis of the project in Chapter 4, however, revealed that efficient knowledge transfer did not take place initially. Domain knowledge was only transferred *reactively* and not *proactively*. What helped in the case under study to resolve this conflict was to explicitly foster the domain knowledge exchange by facilitating the transfer of domain knowledge from experts to the main developer.

During the last part of the project, regular meetings were held within the frame of the Scrum process. Those meetings were scheduled on a regular basis and provided a recurring opportunity for the MD to present the current state of the developed software and to signal the difficulties he currently faced. The other participants and especially the two experts could then propose their help and schedule meetings with the MD in which the resolution of the problems would be discussed. This fostered the domain knowledge integration through direct communication (see also 2.2.3.2).

This successful approach cannot be easily generalized. The essential aspect was to provide an opportunity for proactive knowledge exchange. In the current case, the regular short-meetings with the participation of all domain knowledge experts seemed to provide a good solution under the given circumstances. In this case, the opportunity for proactive knowledge exchange came under the heading of the agile Scrum process. Any other introduction of similar “good practices” could have provided the same benefits. Nevertheless, it is also addressed in the literature that agile approaches such as Scrum are well suited to promote knowledge transfer through socialization. Chau, Maurer, and Melnik [32] suggest that agile

development approaches favor domain knowledge exchange through direct face-to-face communication.

6 CONCLUSION

This work reports on an empirical case study about a software development project conducted at CERN. It highlights the specific context and environment in which the project took place and reveals the associated influence factors that led to diverse issues during the course of the project. Besides this analysis, the work describes the software development carried out by the author in his role as the main software developer of the project. In this way, it offers technical details about the software and hardware infrastructure of CERN's control systems and especially about the system concerned by the project. This substantially complements the collected data and the analysis results of the case study.

To start with, this work described the case study methodology employed and introduced the research topics of the study along with a literature overview. Then, this work described extensively the organizational as well as the technical context in which the project took place. Additionally, the elaborated software solution was presented. The main characteristics of the context were the large amount of specific domain knowledge necessary to successfully develop the software in the given setting.

Further on, this work presented the data collected during the project. Based on this data, the issues that occurred in the frame of the specific context were discussed. The main findings were the difficulties that the main developer faced because of poor domain knowledge integration during the project. These difficulties related mostly to doing requirements as a prerequisite for further development steps. Notably, two communication breakdowns and a failure of the initial document-driven approach were identified and discussed. The solutions that solved those issues were pointed out. They mainly consisted of abandoning a document-driven development approach that apparently did not fit into the given context. An iterative incremental approach using evolutionary prototypes was found to be better suited under the given context. It was shown that those issues and their counteracting measures divided the project into three parts characterized by three different development approaches.

Finally, based on the findings of the case study, generalized lessons learned were discussed. These lessons learned relate to the “people oriented” factors in which the case study's research topics were rooted. They essentially suggest possible approaches when facing a high level of specific domain knowledge. In this frame, the findings of the study were also compared to findings published in literature.

The relationship between domain knowledge and requirements, which was especially prominent in the case under study, suggests a deeper and more faceted relation than discussed until now in the literature. Further theoretical work on this topic has already been started by the author in the frame of the elaboration of this retrospective study [85].

Index of Figures

References

[1] B. Adelson and E. Soloway, "The Role of Domain Experience in Software Design," *IEEE Transactions on Software Engineering*, vol. 11, no. 11, pp. 1351-1360, 1985.

[2] A. Al-rawas and S. Easterbrook, "Communication problems in requirements engineering: A field study," in *Proceedings of the First Westminster Conference on Professional Awareness in Software Engineering*, London, 1996, pp. 47-60.

[3] M. Arruat et al., "Front-End Software Architecture," in *Proceedings of the 11th International Conference on Accelerator and Large Experimental Physics Control Systems*, Knoxville, Tennessee, USA, 2007, pp. 310-312.

[4] M. E. Atwood et al., "Facilitating communication in software development," in *Proceedings of the 1st conference on Designing interactive systems, ACM*, 1995, pp. 65-73.

[5] V. Baggiolini et al., "Remote device access in the new accelerator controls middleware," in *Proceedings of the 8th International Conference on Accelerator and Large Experimental Physics Control Systems*, San Jose, California, 2001, pp. 496-498.

[6] V. Basili, "The role of experimentation in software engineering: past, current, and future," in *Proceedings of the 18th International Conference on Software Engineering*, 1996, pp. 442-449.

[7] V. Basili and A. Turner, "Iterative Enhancement: A Practical Technique for Software Development," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 4, pp. 390-396, 1975.

[8] J. C. Bau, G. Daems, J. Lewis, and J. Philippe, "Managing the real-time behaviour of a particle beam factory: the CERN Proton Synchrotron complex and its timing system principles," *IEEE Transactions on Nuclear Science*, vol. 45, pp. 2004-2007, 1998.

[9] B. Beauté, "How CERN ended up in Geneva," *Refelx*, no. 4, pp. 66-69, 2008.

[10] K. Bennett, "Legacy systems: coping with stress," *IEEE Software*, vol. 12, no. 1, pp. 19-23, 1995.

[11] T. Berners-Lee. Information Management: A Proposal. [Online]. http://www.w3.org/History/1989/proposal.html

[12] A. Beuret et al., "The LHC Lead Injector Chain," in *Proceedins of the 9th European Particle*

Accelerator Conference (EPAC'04), Lucerne, Switzerland, 2004, pp. 1153-1155.

[13] A. Birk, T. Dingsoyr, and T. Stalhane, "Postmortem: never leave a project without it," *IEEE Software*, vol. 19, pp. 43-45, 2002.

[14] D. Bjerklie, "Monsieur WWW," *Reflex*, no. 4, pp. 64-65, 2008.

[15] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, pp. 61-72, 1988.

[16] B. Boehm, "A view of 20th and 21st century software engineering," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 12-29.

[17] B. Boehm, "Anchoring the software process," *IEEE Software*, vol. 13, pp. 73-82, 1996.

[18] B. Boehm et al., "Using the WinWin spiral model: a case study," *Computer*, vol. 31, pp. 33-44, 1998.

[19] J. Boillot, G. Daems, P. Heymans, and M. Overington, "Pulse-To Pulse Modulation of the Beam Characteristics and Utilization in the CERN PS Accelerator Complex," *IEEE Transactions on Nuclear Science*, no. 28, pp. 2195-2197, 1981.

[20] J. Broere, I. Kozsar, R. Garoby, A. Rohlev, and J. Serrano, "All Digital IQ Servo-System for CERN Linacs," in *Proceedings of the 9th European Particle Accelerator Conference (EPAC'04)*, Lucerne, Switzerland, 2004, pp. 605-607.

[21] F. Brooks, "No Silver Bullet Essence and Accidents of Software Engineering," *Computer*, vol. 20, pp. 10-19, 1987.

[22] F. Brooks, *The mythical man-month (anniversary ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., 1995.

[23] A. Butterworth, J. Molendijk, R. Sorokoletov, and F. Weierud, "Control of the Low Level RF system of the Large Hadron Collider," in *Proceedings of the 10th International Conference on Accelerator and Large Experimental Physics Control Systems*, Geneva, 2005.

[24] CERN, "2010 ion run: completed!," *CERN Bulletin*, no. 50, Dec. 2010.

[25] CERN, "Annual Report," Geneva, 2008.

[26] CERN. (2011, March) Beams Department Mandate. [Online]. https://espace.cern.ch/be-dep/Mandate.aspx

[27] CERN, "CERN Council rings the changes," Geneva, PR23.03, 2003.

[28] CERN, "CERN looks forward to exciting future," Geneva, PR 04.96, 1996.

[29] CERN, "CERN's new organisational structure," *CERN Bulletin*, no. 4, 2009.

[30] CERN, "Forty years of CERN's proton synchrotron," *CERN Courier*, vol. 39, no. 10, pp. 15-18, 1999.

[31] CERN, *Infinitely CERN - Memories of fifty years of research*. Geneva, Switzerland: Editions Suzanne Hurter, 2004.

[32] T. Chau, F. Maurer, and G. Melnik, "Knowledge sharing: agile methods vs. Tayloristic methods," in *Proceedings of the 12th IEEE International Workshops Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2003)*, 2003, pp. 302-307.

[33] A. Cockburn, *Agile software development*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[34] B. Collier, T. DeMarco, and P. Fearey, "A defined process for project post mortem review," *IEEE Software*, vol. 13, pp. 65-72, 1996.

[35] J. L. Connell and L. Brice, "The impact of implementing a rapid prototype on system maintenance," in *Proceedings of the AFIPS, IEEE CS Press*, Los Alamitos, California, 1985, pp. 515-524.

[36] *Convention for the establishment of a European organization for nuclear research*. Paris: CERN, 1953.

[37] J. Coughlan, M. Lycett, and R. D. Macredie, "Communication issues in requirements elicitation: a content analysis of stakeholder experiences," *Information and Software Technology*, vol. 45, pp. 525-537, 2003.

[38] J. Coughlan and R. D. Macredie, "Effective Communication in Requirements Elicitation: A Comparison of Methodologies," *Requirements Engineering*, vol. 7, pp. 47-60, 2002.

[39] G. Cugola and C. Ghezzi, "Software processes: a retrospective and a path to the future," *Software Process: Improvement and Practice*, vol. 4, no. 3, pp. 101-123, 1998.

[40] H. Cunningham, K. Humphreys, R. Gaizauskas, and Y. Wilks, "Software infrastructure for natural language processing," in *Proceedings of the fifth conference on Applied natural language processing*, Morristown, NJ, USA, 1997, pp. 237-244.

[41] J. Cuperus, R. Billen, and M. Lelaizant, "The Configuration Database for the CERN Accelerator Control System," in *9th International Conference on Accelerator and Large Experimental Physics Control Systems*, Gyeongju, Korea, 2003, pp. 309-311.

[42] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, no. 11, pp. 1268-1287, 1988.

[43] A. Daneels, "ICALEPCS 2005: Geneva," *CERN Courier*, vol. 46, no. 2, March 2006.

[44] T. DeMarco and T. Lister, *Peopleware: Productive Projects and T2eams*, 2nd ed. New

York, NY, USA: Dorset House Publishing Co., 1999.

[45] K. C. Desouza, "Barriers to effective use of knowledge management systems in software engineering," *Communication of the ACM*, vol. 46, pp. 99-101, 2003.

[46] K. Ericsson and H. Simon, "Verbal reports on thinking," in *Introspection in Second Language Research*, C. Faerch and G. Kasper, Eds., 1987.

[47] S. R. Faulk, "Software requirements: A tutorial," in *Software Engineering Project Management*, 2nd ed., Richard H. Thayer, Ed.: IEEE Computer Society Press, 1997.

[48] N. Fenton, S. L. Pfleeger, and R. L. Glass, "Science and substance: a challenge to software engineers," *IEEE Software*, vol. 11, pp. 86-95, 1994.

[49] B. Frammery, "The LHC Control System," in *Proceedings of the 10th International Conference on Accelerator and Large Experimental Physics Control Systems*, Geneva, 2005.

[50] A. Fuggetta, "Software Process: A Roadmap," *Proceedings of the Conference on The Future of Software Engineering (ICSE'00)*, pp. 25-34, 2000.

[51] R. Garoby, "Scenarios for upgrading the LHC injectors," in *LHC-LUMI-06 Proceedings*, 2006, pp. 107-110.

[52] A. Guerrero et al., "CERN Front-End Software Architecture for Accelerator Controls," in *Proceedings of the 9th International Conference on Accelerator and Large Experimental Physics Control Systems*, Gyeongju, Korea, 2003, pp. 342-344.

[53] T. Hall, S. Beecham, and A. Rainer, "Requirements problems in twelve software companies: an empirical analysis," *IEE Proceedings Software*, vol. 149, pp. 153-160, 2002.

[54] J. Highsmith and A. Cockburn, "Agile software development: the business of innovation," *Computer*, vol. 34, pp. 120-127, 2001.

[55] J. Highsmith and A. Cockburn, "Agile software development: the business of innovation," *Computer*, vol. 34, pp. 120-122, Sept. 2001.

[56] T. Hoffmann, *FESA Primer – A simple and fast approach from a user's point of view.*: CERN, 2007, Internal Documentation.

[57] M. Host and P. Runeson, "Checklists for Software Engineering Case Study Research," in *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, 2007, pp. 479-481.

[58] W. S. Humphrey, "The software engineering process: definition and scope," in *Proceedings of the 4th international software process workshop on Representing and enacting the software process*, New York, 1988, pp. 82-83.

[59] H. Kaindl and D. Svetinovic, "On confusion between requirements and their representations," *Requirements Engineering*, vol. 15, no. 3, pp. 307-311, 2010.

[60] B. A. Kitchenham et al., "Preliminary guidelines for empirical research in software engineering," *IEEE Transactions on Software Engineering*, vol. 28, pp. 721-734, 2002.

[61] B. Kitchenham, L. Pickard, and S. L. Pfleeger, "Case studies for method and tool evaluation," *IEEE Software*, vol. 12, pp. 52-62, 1995.

[62] K. Kostro, J. Andersson, F. Di Maio, and S. Jensen, "The Controls Middleware (CMW) at CERN Status and Usage," in *Proceedings of the 9th International Conference on Accelerator and Large Experimental Physics Control Systems*, Gyeongju, Korea, 2003, pp. 318-321.

[63] K. Kostro et al., "Controls Middleware - The New Generation," in *Proceedings of the 8th European Particle Accelerator Conference (EPAC)*, Paris, France, 2002, pp. 2028-2030.

[64] R. E. Kraut and L. A. Streeter, "Coordination in software development," *Commun. ACM*, vol. 38, pp. 69-81, 1995.

[65] P. Kruchten and C. Thompson, "Iterative software development for large Ada programs," *Reliable Software Technologies - Ada-Europe '96*, vol. 1088, pp. 101-110, 1996.

[66] G. Kruk and M. Peryt, "JDataViewer–JAVA-Based Charting Library," in *Proceedings of the 12th International Conference on Accelerator and Large Experimental Control Systems*, Kobe, Japan, 2009, pp. 856-858.

[67] M. Lamont and L. Mestre, "LHC Era Core Control Application Software," in *Proceedings of the 10th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2005.

[68] C. Larman and V. R. Basili, "Iterative and incremental developments. A brief history," *Computer*, vol. 36, no. 6, pp. 47-56, 2003.

[69] J. Lewis et al., "The CERN LHC Central timing, a vertical slice," in *Proceedings of the 11th International Conference on Accelerator and Large Experimental Control Systems*, Knoxville, Tennessee, 2007.

[70] J. Lewis, J. C. Bau, and M. Jonker, "The Central Beam and Cycle Management of the CERN Accelerator Complex," in *Proceedings of the 7th International Conference on Accelerator and Large Experimental Physics Control Systems*, Trieste, Italy, 1999, pp. 14-16.

[71] K. R. Linberg, "Software developer perceptions about software project failure: a case study," *The Journal of Systems & Software*, vol. 49, pp. 177-192, 1999.

[72] M. Loy and R. Eckstein, *Java Swing*, 2nd ed.: O'Reilly Media, Inc., 2002.

[73] D. Manglunki, "The CERN Control Centre: Setting Standards for the 21st Century," in *Proceedings of the 11th International Conference on Accelerator and Large Experimental Control Systems*, Knoxville, Tennessee, 2007, pp. 603-605.

[74] M. B. Miles and A. M. Huberman, "Early Steps in Analysis," in *Qualitative data analysis: an expanded sourcebook*, 2nd ed.: Sage Publications, 1994, pp. 50-89.

[75] H. W. Miller, *Reengineering legacy software systems*. Newton, MA, USA: Digital Press, 1998.

[76] R. Morien and P. Wongthongtham, "Supporting agility in software development projects - defining a project ontology," in *Proceedings of the 2nd IEEE International Conference Digital Ecosystems and Technologies (DEST 2008)*, 2008, pp. 229-234.

[77] Bashar Nuseibeh and Steve Easterbrook, "Requirements engineering: a roadmap," in *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, 2000, pp. 35-46.

[78] G. M. Olson, J. S. Olson, M. R. Carter, and M. Storrosten, "Small group design meetings: an analysis of collaboration," *Human-Computer Interaction*, vol. 7, pp. 347-374, 1992.

[79] D. L. Parnas, *Software fundamentals: collected papers by David L. Parnas*, D. Hoffman and D. Weiss, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[80] D. L. Parnas and P. C. Clements, "A rational design process: How and why to fake it," *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 251-257, 1986.

[81] R. R. Patrashkova-Volzdoska, S. A. McComb, S. G. Green, and W. D. Compton, "Examining a curvilinear relationship between communication frequency and team performance in cross-functional project teams," *IEEE Transactions on Engineering Management*, vol. 50, pp. 262-269, 2003.

[82] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: a roadmap," in *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*, 2000, pp. 345-355.

[83] D. E. Perry, S. E. Sim, and S. Easterbrook, "Case Studies for Software Engineers (Tutorial)," in *Proceeding of the 28th International Conference on Software Engineering*, 2006, pp. 1045-1046.

[84] D. E. Perry, N. A. Staudenmayer, and L. G. Votta, "People, organizations, and process improvement," *IEEE Software*, vol. 11, pp. 36-45, 1994.

[85] L. Pilat and H. Kaindl, "A Knowledge Management Perspective of Requirements Engineering," in *Proceedings of the Fifth IEEE International Conference on Research Challenges in Information Science*, 2011, in print.

[86] C. Potts, "Software-engineering research revisited," *IEEE Software*, vol. 10, pp. 19-28, 1993.

[87] C. Potts and L. Catledge, "Collaborative conceptual design: A large software project case study," *Computer Supported Cooperative Work (CSCW)*, vol. 5, pp. 415-445, 1996.

[88] J. D. Procaccino, J. M. Verner, S. P. Overmyer, and M. E. Darter, "Case study: factors for early prediction of software development success," *Information and Software technology*, vol. 44, pp. 53-62, 2002.

[89] B. Ramesh, T. Powers, C. Stubbs, and M. Edwards, "Implementing requirements traceability: a case study," in *Second IEEE International Symposium on Requirements Engineering*, 1995, pp. 89-95.

[90] L. Rising and N. S. Janoff, "The Scrum Software Development Process for Small Teams," *IEEE Software*, vol. 17, no. 4, pp. 26-32, July/August 2000.

[91] P. N. Robillard, "The role of knowledge in software development," *Communications of the ACM*, vol. 42, no. 1, pp. 87-92, 1999.

[92] R. N. Robillard, "The role of knowledge in software development," *Communications of the ACM*, vol. 42, pp. 87-92, 1999.

[93] A. Rohlev, J. Serrano, and R. Garoby, "All digital IQ servo-system for CERN linacs," in *Proceedings of the Particle Accelerator Conference (PAC)*, vol. 4, 2003.

[94] L. Rossi, "The Large Hadron Collider and the role of superconductivity in one of the largest scientific enterprises," *IEEE Transactions on Applied Superconductivity*, no. 17, pp. 1005-1014, 2007.

[95] P. A. Roussel, K. N. Saad, and T. J. Erickson, *Third Generation R & D: Managing the link to corporate strategy*.: Harvard Business School Press, 1991.

[96] W. Royce, "Managing the Development of Large Software Systems 1970," in *Proceedings, IEEE WESCON*, 1970, pp. 1-9.

[97] I. Rus and M. Lindvall, "Knowledge management in software engineering," *IEEE Software*, vol. 19, pp. 26-38, 2002.

[98] D. Saraga, "The extraordinary quest for our origins," *Reflex*, no. 4, pp. 38-46, 2008.

[99] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, pp. 557-572, 1999.

[100] J. Serrano, P. Alvarez, D. Dominguez, and J. Lewis, "Nanosecond Level UTC Timing Generation and Staming in CERN's LHC," in *Proceedings of the 9th International Conference on Accelerator and Large Experimental Control Systems*, Gyeongju, Korea, 2003, pp. 119-121.

[101] J. Serrano, A. Rohlev, and I. Kozsar, "FPGA-Based Low Level Control of CERN's LINAC 3 Cavities," in *Proceedings of the 10th International Conference on Accelerator and Large Experimental Physics Control Systems*, Geneva, 2005.

[102] W. Sliwinski and N. Stapley, "Applying Agile Project Management for Accelerator Controls Software," in *Proceedings of the 11th International Conference on Accelerator and Large Experimental Control Systems*, Knoxville, Tennessee, USA, 2007, pp. 612-614.

[103] R. E. Stake, *The art of case study research*.: Sage Publications, 1995, book.

[104] T. Stalhane, T. Dingsoyr, G. Hanssen, and N. Moe, "Post Mortem - An Assessment of Two Approaches," in *Empirical Methods and Studies in Software Engineering*.: Springer Berlin / Heidelberg, 2003, pp. 129-141.

[105] J. Stempfle and P. Badke-Schaub, "Thinking in design teams - an analysis of team communication," *Design Studies*, vol. 23, pp. 473-496, 2002.

[106] L. G. Terveen, P. G. Selfridge, and M. D. Long, "From “folklore” to “living design memory”," in *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, Amsterdam, 1993, pp. 15-22.

[107] A. Tiwana, "An empirical study of the effect of knowledge integration on software development performance," *Information and Software Technology*, vol. 46, pp. 899-906, 2004.

[108] L. G. Votta, "By the Way, Has Anyone Studied Any Real Programmers, Yet?," in *Proceedings of the 9th International Software Process Workshop*, 1994, pp. 93-95.

[109] D. B. Walz, J. J. Elam, and B. Curtis, "Inside a software design team: knowledge acquisition, sharing, and integration," *Communications of the ACM*, vol. 36, no. 10, pp. 63-77, 1993.

[110] C. Wohlin, M. Höst, and K. Henningsson, "Empirical Research Methods in Software Engineering," *Empirical Methods and Studies in Software Engineering*, vol. 2765/2003, pp. 7-23, 2003.

[111] R. K. Yin, *Case Study Research*, Third Edition ed.: Sage Publications, 2003.

[112] R. K. Yin, "The Case Study Crisis: Some Answers," *Administrative Science Quarterly*, vol. 26, pp. 58-65, 1981.